MODULAR FPGA SYSTEMS WITH SUPPORT FOR DYNAMIC WORKLOADS AND VIRTUALISATION

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2020

Anuj A Vaishnav

School of Engineering
Department of Computer Science

Contents

Word Count: 35646

List of Tables

List of Figures

Abstract

This thesis shows that it is feasible to build modular FPGA systems which can dynamically change the hardware resources in the spatial and the temporal domains using existing tools and accelerators, to improve maintainability, adaptability, and accessibility for FPGA systems.

To achieve this, first, a modular FPGA development flow is proposed to build an FPGA operating system, spanning both software and hardware level, where each component can be changed, reused, or ported to other systems with minor changes and recompilation steps. This modular flow removes the dependencies in existing infrastructures while supporting a wide range of FPGA use cases (static acceleration to dynamic multi-tenant acceleration) with easy-to-use software interfaces for both software and hardware developers. This platform is then extended using OpenCL to perform scheduling across two types of computing devices available on modern FPGA systems, i.e. both CPU and FPGA. To improve the adaptability of the system while maximising system utilisation, a novel concept called 'resource elasticity' is proposed to allow the system to change the amount of resources used by a task transparently from the user in both space and time domain. Further, this is combined with the ability to dynamically move computation between CPU and FPGA (as well as collaborative execution) to allow changing the 1) device type (CPU or FPGA or both), 2) accelerator type, 3) number of compute units and 4) workload partitioning while supporting multi-tasking. Finally, this platform is used with multiple nodes to show how we can perform live migration across different FPGA nodes to allow system maintenance, load balancing and fault tolerance at the cluster level.

Overall, with these contributions, this thesis enables building scalable FPGA systems that can support environments with dynamic workloads (e.g. cloud and edge computing) without compromising on the programmability, performance and ease of use for FPGAs. Further, the proposed solutions also improve the productivity for general FPGA users through a modular development flow and high-level user interfaces.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

'For my father, who taught me to have faith in myself'.

Acknowledgements

First of all, I would like to whole-heartedly thank and appreciate my supervisor, Dr Dirk Koch, for his guidance, support, and motivation. His open-door policy, impromptu discussions, and careful criticism of all my research work has taught me much more than I had imagined, and this has helped me become a better thinker and engineer. Without him, this thesis would not have been possible.

I also wish to express my deepest gratitude towards my co-supervisor, Dr James Garside, for his guidance and proofreading my research papers. The insights he shared have often brought clarity during times of ambiguity and uncertainty and help me plan the major milestones better.

Further, I would like to thank my colleague, Khoa Dang Pham, who initially showed me the ropes of tools and academia in general as a mentor and later became a great collaborator for my research and more importantly, a good friend. His contribution to my learning and research achievements has been monumental.

I would also like to thank Kristiyan Manev for stimulating debates and collaborations, Joseph Powell for helping with implementation and debugging during his internship, and rest of the Advanced Processor and Technology (APT) group for creating a productive and enjoyable environment. Their enthusiasm and helpful nature has made my time at the university much more exciting and engaging.

Lastly, I am deeply grateful to my family and friends for their constant support, patience and motivation throughout the past three years. They kept me going on, and this work would not have been possible without them.

Chapter 1

Need for Maintainability, Adaptability and Accessibility

Thesis statement: It is feasible to build modular FPGA systems that can dynamically change the hardware resources in the spatial and the temporal domains for performance improvement using existing tools and accelerators if we adopt right abstraction layers and programming models.

1.1 Motivation

With the slowdown in the transistor scaling, adding more transistors on a chip to improve performance using general-purpose CPU architecture and multi-processing is becoming difficult because of lower manufacturing yield [29]. Hence, industry and academia are considering an alternate form of computation which specialises towards their target application needs [42]. One way to achieve this is by producing application-specific chips (ASICs). However, fabricating a chip is expensive and only suitable for the widely used application. More importantly, once fabricated, the chip cannot easily adapt to new algorithms or market needs. An alternative to this are Field Programmable Gate Arrays (FPGAs). They provide a trade-off between cost and specialisation, as they can be manufactured once but configured to application-specific needs even during system operation. This allows FPGAs to accelerate a broader range of applications using existing fabrication technology than standard ASICs. In particular, their ability for customised computing provides an advantage over standard CPUs not just in terms of performance but also higher energy efficiency for many applications [66, 88]. This combination of performance and energy efficiency makes them a suitable candidate

for computing at the edge and exascale computing systems where energy consumption restrictions are stringent, and applications require high performance [8, 74].

In particular, these advantages have brought the FPGAs in cloud datacenters [88, 92] and at the edge [8, 133], making them accessible to the masses at an unprecedented scale. To make FPGAs easier to use and manage, they are virtualised with an 'FPGA shell' to host multiple hardware accelerators. An FPGA shell is a hardware infrastructure comprising the standard functionalities required for acceleration on FPGAs, such as memory and network access, and the ability to host multiple accelerators. The accelerators for these shells can be described directly as hardware circuits in hardware description languages (HDL) or in high-level software languages, such as C or OpenCL, with the help of high-level synthesis (HLS) [56, 119]. To abstract them further, cloud providers and FPGA vendors provide software libraries accessible by users which leverage these accelerators behind the scenes for high performance while providing the same interface as standard software systems [130, 131, 133]. This mode of use is known as Acceleration-as-a-Service (AaaS). It allows software developers and application-domain users to benefit from hardware acceleration without in-depth knowledge of FPGAs.

Overall, as a result, the FPGA ecosystem is moving towards a lightweight FPGA operating system (OS) which can provide a standard set of abstraction layers and common system functionalities to build scalable systems required to support large-scale deployment and a variety of user needs, i.e. help achieve FPGA virtualisation [108] (see Section 2.6 for details). Ideally, this FPGA OS would also allow to share the FPGA transparently between multiple users and adapt to changing workload requirements to maximise the system utilisation like a software OS does for CPUs. This is crucial for cloud infrastructures as this would allow to serve more users with the same number of FPGA nodes and adapt to changing user needs. Further, the OS needs to ensure an easy-to-use interface for *both* software and hardware developers, i.e. provide high-level acceleration libraries but also allow easy integration for new accelerators built by hardware developers.

However, achieving this is difficult as each FPGA and the accelerator is different and requires a different set of settings before they can deliver high performance and energy efficiency (see Figure 1.1). In particular, the current FPGA development flow passes these differences between component compilations which lead to recompilation of the entire system for minor changes, for example, when changing EDA tool version or adding new IP in the shell [129] (as shown in Figure 1.2). Hence, we must first solve

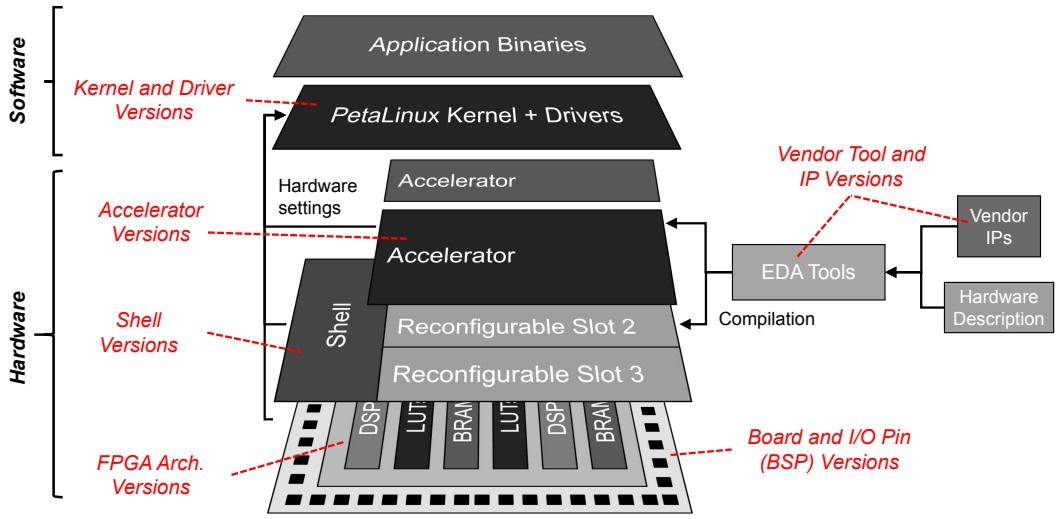


Figure 1.1: Variety of sources for heterogeneity in the FPGA development and deployment stack span across core elements of both hardware and software.

the problem of handling heterogeneity in the FPGA stack with lightweight abstraction layers to improve *maintainability* and design flow scalability. Past projects such as ReconOS [67] and Borph [99] aimed to solve some of these issues via abstraction interfaces such as threads or UNIX processes which allowed hardware accelerators to interact and integrate directly with software operating systems like software processes. However, this only solves the issue for interaction between hardware and software layers, and not between the hardware components (shell and accelerators). Further, in contrast to their philosophy, current FPGA systems aim towards enabling hardware acceleration for software applications which allows them to avoid expensive primitives required for thread synchronisation [67] and process communication [99] at the hardware level.

To allow FPGA systems to *adapt* to dynamic workloads, we need to be able to pause and resume accelerators with low overhead. However, this is very different from CPUs, where we can save and restore registers for context-switch in software. In contrast, FPGA accelerators are hardware circuits and can store their internal state in an arbitrary amount of registers and block RAMs available on-chip. In the past, the preemptive context-switch for FPGA accelerators has been proposed using configuration read-back and scan chains to read the internal state [39, 58, 76, 98] (see Section 3.4.2 for details). However, these approaches impose expensive context-switch penalties

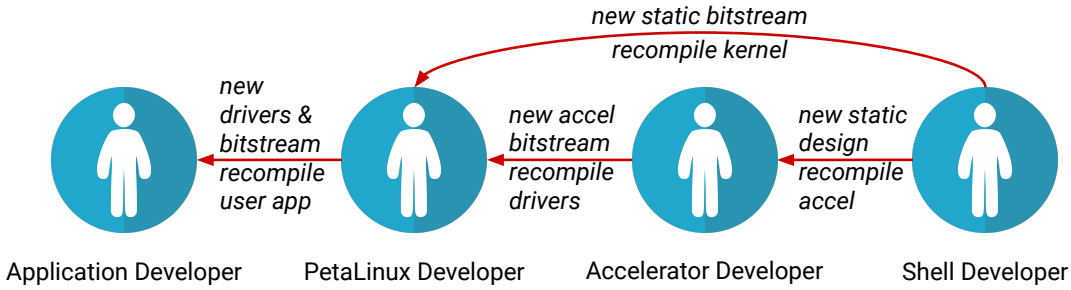


Figure 1.2: If we update the shell IP or EDA tool version, the rest of the system components must be recompiled because of the dependencies in standard tool flow [128, 129].

and restriction on accelerator design, which reduces the acceleration benefits of FPGAs. Consequently, current FPGA systems often employ a run-to-completion model and refrain from runtime resource reallocation entirely [56, 119]. This results in low utilisation and long wait-times for acceleration requests (see Section 5.3). Moreover, this restriction at node level[1] also reduces the *adaptability* at cluster level as the cluster manager cannot move computation from one FPGA node to another for maintenance, load balancing or fault tolerance purposes until the application completes.

To provide better *accessibility* for both software and hardware developers, we need to ensure that high-level interfaces are similar to existing software stacks and easy to extend to new accelerators without imposing design constraints or extensive effort. Currently, the library interfaces are easy to use for software programmers [123, 130, 131] but require an additional effort from the hardware designers for writing new hardware interface wrappers or drivers and their integration into software stack [119, 128] (see Section 3.3).

1.2 Contribution

To solve these challenges of maintainability, adaptability, and accessibility (MAA), in this thesis, we propose:

- **A modular FPGA operating system (FOS)**: FOS adopts a modular FPGA development flow to allow each type of OS component (hardware or software) to be replaced, reused, or ported to different FPGA systems without extensive design effort or compilation time. Moreover, the runtime system provides full support

[1] A single FPGA chip.

for multi-tenancy with the ability to execute accelerators written in various languages (C, C++, OpenCL, or RTL) concurrently and dynamically replicate or switch to a different version of an accelerator to improve utilisation and system performance. To interact with these accelerators, application developers can use high-level APIs (available in multiple languages) to access the FPGA in three modes: 1) static acceleration for a single user, 2) dynamic acceleration for a single user and 3) dynamic acceleration in multi-tenant environments. At the same time, hardware developers can write light-weight interface descriptions to integrate their hardware accelerators into the FOS platform and benefit from its high-level software APIs.

- **Heterogeneous resource elastic scheduling**: Resource elasticity for FPGAs aims to maximise the resource utilisation and performance by dynamically changing the spatial and temporal resource allocation of FPGA accelerators. To achieve this, we adopt cooperative context-switching based on data parallelism and scale the resource allocation by changing to an implementation alternative or replication transparently from the user. Since modern FPGA systems often include a CPU on board (soft or hard) [126], our runtime system also scales the i) number of CPU cores, ii) moves computation between CPU and FPGA, and iii) performs collaborative execution[2] to maximise the system performance without user involvement when using OpenCL[3]. From an application's point of view, the runtime system exposes only a generic accelerator call and transparently performs the entire resource allocation and hardware accelerator management (including FPGA configuration).

- **Live migration for OpenCL FPGA accelerators**: Using FOS and the resource elastic runtime system, we can dynamically change resource allocation on a single FPGA node. To extend this towards an FPGA cluster, we propose blocking and non-blocking live migration methods for OpenCL FPGA accelerators with minimal overhead to improve the flexibility of a cluster manager. We demonstrate how we can use this to transparently i) update FPGA nodes, ii) improve load balancing, and iii) achieve better reliability.

[2]Execute an application on both CPU and FPGA together.

[3]For FPGA only systems, resource elasticity also extends to C, C++ and RTL accelerators via FOS.

1.3 Scope

The primary contribution of this thesis lies in the proposed abstractions layers and the resource elastic management for FPGAs. We use existing tools and standard development practices wherever possible to keep the solutions applicable to existing systems.

In particular, we perform the modifications required in the partial reconfiguration (PR) flow to reduce hardware dependencies using the philosophy of [60][4] and using open-source research tools available [6, 84, 111]. Hence, the PR flow is not a contribution of this thesis, but only its application to build modular FPGA systems.

To build a multi-node FPGA system, we need to consider many aspects ranging from network connectivity to distributed frameworks to fault-tolerance mechanisms. However, in this thesis, we only focus on how we can use the proposed solution to better support the goals and needs of multi-node systems. For more details on the problems and solutions proposed in the research community, refer to Section 2.5.

Further, the methods and concepts proposed in this thesis can be adopted with other virtualisation technologies such as software virtual machines monitors (VMM) and FPGA overlays (see Section 2.6). However, their integration and optimisation for low overhead require solving different research problems and hence lies out of the scope of this thesis.

Additionally, the focus of this thesis is not on how we can automatically develop high-performance FPGA accelerators of varying resource requirements used for resource elastic scheduling. We believe design space exploration for resource budgeting to be a related but separate problem from runtime resource allocation.

1.4 Thesis Outline

The rest of this document is organised into five main chapters:

- **Chapter 2:** Describes and categorises the different types of FPGA virtualisation and how we can combine them to provide a more comprehensive set of features. Moreover, it identifies the issues and the gap which still exist in the current FPGA ecosystem towards maintainability, adaptability, and accessibility.

- **Chapter 3:** States the proposed solutions to the issues discussed in Chapter 2

[4] Prohibit all wires crossing between static and reconfigurable region except for pre-defined interfaces during place and route stage. This allows us to compile static logic and accelerators separately.

(modular development flow, resource elastic scheduling, and transparent live migration) and describes the background definitions, core concepts and methods involved in these solutions. Further, it discusses the trade-offs involved in the scheduling problem and various methods we can adopt to achieve our aim of building modular FPGA systems with spatial resource sharing. Finally, it concludes with a summary of the methods adopted in this thesis and their system requirements.

- **Chapter 4:** Describes and discusses the implementation details to realise the methods and concepts proposed in Chapter 3 in three parts. First, it describes FOS — the FPGA operating system infrastructure to support modularity in the FPGA development flow and FPGA runtime management. Then it presents the details of the heterogeneous runtime system which runs on top of FOS shell and driver layers to schedule resources on both CPU and FPGA transparently. Followed by the multi-node architecture to enable and implement OpenCL accelerator migration methods from Chapter 3.

- **Chapter 5:** Evaluates the methods and implementations based on their performance and overhead under different use-case scenarios, workload behaviours and FPGA platforms using latency, throughput, wait-time, schedule makespan (completion-time) and system utilisation as primary metrics. This evaluation is performed in three phases: we first evaluate the base modular FPGA infrastructure, then the dynamic resource allocation on CPU+FPGA systems and the accelerator migration across FPGA nodes.

- **Chapter 6:** Summarises and states the key contributions of this thesis and their implications for future FPGA systems. It also identifies the future work required to further improve the efficacy of proposed methods and the potential new research directions.

The appendices of this document include 1) the theory behind resource elastic scheduling for FPGAs and heterogeneous systems in Appendix A, and 2) a brute-force approach to resource elastic scheduling for fairness and performance and its evaluation on long and short running task workload in Appendix B.

1.5 Publications

Following papers have contributed towards this thesis:

1. **A. Vaishnav**, K. D. Pham, and D. Koch. A Survey on FPGA Virtualization. In *International Conference on Field Programmable Logic and Applications (FPL)*, 2018.

2. **A. Vaishnav**, K. D. Pham, D. Koch, and J. Garside. Resource Elastic Virtualization for FPGAs Using OpenCL. In *International Conference on Field Programmable Logic and Applications (FPL)*, 2018.

3. K. D. Pham, **A. Vaishnav**, M. Vesper, and D. Koch. ZUCL: A ZYNQ UltraScale+ Framework for OpenCL HLS Applications. In *International Workshop on FPGAs for Software Programmers (FSP)*, 2018.

4. **A. Vaishnav**, K. Pham, and D. Koch. Live Migration for OpenCL FPGA Accelerators. In *International Conference on Field-Programmable Technology (FPT)*, 2018.

5. **A. Vaishnav**, K. D. Pham, and D. Koch. Heterogeneous Resource-Elastic Scheduling for CPU+FPGA Architectures. In *International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART)*, 2019.

6. **A. Vaishnav**, K. D. Pham, K. Manev, and D. Koch. The FOS (FPGA Operating System) Demo. In *International Conference on Field Programmable Logic and Applications (FPL)*, 2019.

7. K. D. Pham, K. Paraskevas, **A. Vaishnav**, A. Attwood, M. Vesper, and D. Koch. ZUCL 2.0: Virtualised Memory and Communication for ZYNQ UltraScale+ FPGAs. In *International Workshop on FPGAs for Software Programmers (FSP)*, 2019.

8. K. Manev, **A. Vaishnav**, and D. Koch. Unexpected Diversity: Quantitative Memory Analysis for Zynq UltraScale+ Systems. In *International Conference on Field-Programmable Technology (FPT)*, 2019. **(Co-first author)**

9. **A. Vaishnav**, K. D. Pham, J. Powell and D. Koch. FOS: A Modular FPGA Operating System for Dynamic Workloads. In *ACM Transaction Reconfigurable Technology (TRETS)*, 2020.

Additional papers and contributions during the span of the PhD which are not part of this thesis:

- K. Pham, E. Horta, D. Koch, **A. Vaishnav**, and T. Kuhn. IPRDF: An Isolated Partial Reconfiguration Design Flow for Xilinx FPGAs. In *12th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, 2018.
- K. Manev, **A. Vaishnav**, C. Kritikakis, and D. Koch. Scalable Filtering Modules for Database Acceleration on FPGAs. In *10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART)*, 2019.

Chapter 2

Survey on FPGA Virtualisation

2.1 Chapter Overview

This chapter answers the question of what are the existing ways to abstract and manage the FPGA resources and heterogeneity. To do this, a survey on the state-of-the-art methods and their aims for FPGA virtualisation at all levels of FPGA stack (from FPGA architecture to FPGA clusters) is presented with a discussion on how these methods combine to form the current FPGA ecosystem. In particular, this survey reveals the three critical areas in which present methods for FPGA virtualisation do not well match the industry needs and serves as the motivation for this thesis.

2.2 Understanding FPGA Virtualisation

The term "virtualisation" has acquired various meanings over time, with a characteristic trait being the introduction of an abstract layer to simplify the interface and hide the system complexity. In particular, for FPGA virtualisation, the definitions and techniques have changed over time due to changes in application requirements when compared to an earlier survey on FPGA virtualisation in 2004 by Plessl and Platzner [87]. In that work, FPGA virtualisation was classified into three categories: temporal partitioning, virtualised execution, and virtual machine.

Temporal partitioning is used to fit large designs on relatively smaller FPGAs by reconfiguring an FPGA to host a partition of a design one after other at a time. Temporal partitioning is still in use for certain applications like ASIC emulation. However, majority of the current applications which require more FPGA resources than available on a single chip also tend to require executing the application in parallel, i.e. in

the spatial domain with multiple FPGAs rather than in time [82, 88]. Thus, temporal partitioning is nowadays usually used at task-level for large-scale datacenters where a task may span multiple FPGAs but may be swapped with another task in time.

Virtualized execution in the survey [87] was used to define the approach of splitting up applications into multiple communicating tasks (e.g., following a Petri-net model) and using a runtime system to manage them. The aim of this was to support device independence within a device family using dynamic reallocation and bitstream relocation. However, in the recent years, such runtime systems are deployed to support a wide range of applications while separating application functionality from standard static functionalities (such as I/O and communication). Hence, it is now used not only for device independence within the family but also for higher design productivity, isolation and resource management. The static part which provides support for accelerators is often referred to as a shell [88] or Hypervisor for virtual FPGAs (vFPGAs) [59] and details of this will be discussed in Section 2.4.

Finally, Plessl and Platzner defined *virtual machines* to be systems which provide complete device independence by using an abstract architecture to describe applications. This architecture could be translated later into native architecture by a remapping tool or an interpreter (i.e. an FPGA hosting the virtual machine). This approach, in particular, is now known as *FPGA overlays* [100] (also called Intermediate architecture or Intermediate fabric [22]) where the abstract architecture can be defined in many ways as discussed further in Section 2.3.1.

Nowadays, FPGA virtualisation is starting to coincide with techniques for software virtualisation at a conceptual level, with growing support for heterogeneous systems and concepts such as Acceleration-as-a-Service (AaaS) [31].

The objectives of FPGA virtualisation are similar to the core objectives that resulted in the development of virtualisation used in traditional CPU/software systems. The main objectives are:

- **Multi-tenancy:** Ability to serve multiple different users using the same FPGA fabric.
- **Resource Management:** Providing an abstraction/driver layer to the FPGA fabric and means of scheduling tasks to the FPGA as well as monitoring its resource usage.
- **Flexibility:** Ability to support a wide range of acceleration workload, i.e. from custom accelerators to framework-specific accelerators to a bump in the wire

model (i.e. transparent in-network processing).

- **Isolation:** Providing the illusion of being a sole user of the FPGA resources for better security, fewer dependencies and correctness of the program execution.
- **Scalability:** The system/application can scale to multiple different FPGAs or can support multiple different users at relatively low overhead.
- **Performance:** The impact of virtualisation should be minimal on the performance achievable and the FPGA resources usable by the user application.
- **Security:** Ensuring that information between tenants is not leaked and for safekeeping the infrastructure from malicious users.
- **Resilience:** Ability to keep the system/service running despite failures.
- **Design productivity:** Improving the time to market and reducing the complexity of deploying designs on an FPGA platform.

Despite sharing these objectives, virtualisation techniques from the software domain cannot always straightforwardly be applied to FPGAs. This is mostly due to one fundamental difference between CPU/GPUs and FPGAs: applications are hardware circuits rather than a set of (machine) instructions. This leads to differences which require consideration when devising a solution for virtualisation. These differences include a very high context switch penalty, space-time sharing rather than just time, different development tools, high development time, and high heterogeneity in the system as each accelerator represents a distinct hardware module. Thus, many different FPGA virtualisation approaches have been proposed as summarised in Figure 2.1.

2.2.1 Classification of FPGA Virtualisation

Existing work on FPGA virtualisation can be classified in many ways, for example, based on the virtualisation techniques, different use cases or execution models. However, these classification criteria would include a time context, and the classification may change, for example, with the advent of new applications or requirements over time. Instead, we propose a classification scheme based on the abstraction levels of the computational systems that apply virtualisation, such that same classification can capture future changes in the field and categorise the work at following levels (as shown in Figure 2.1): Multi-node level, Node level, and Resource level.

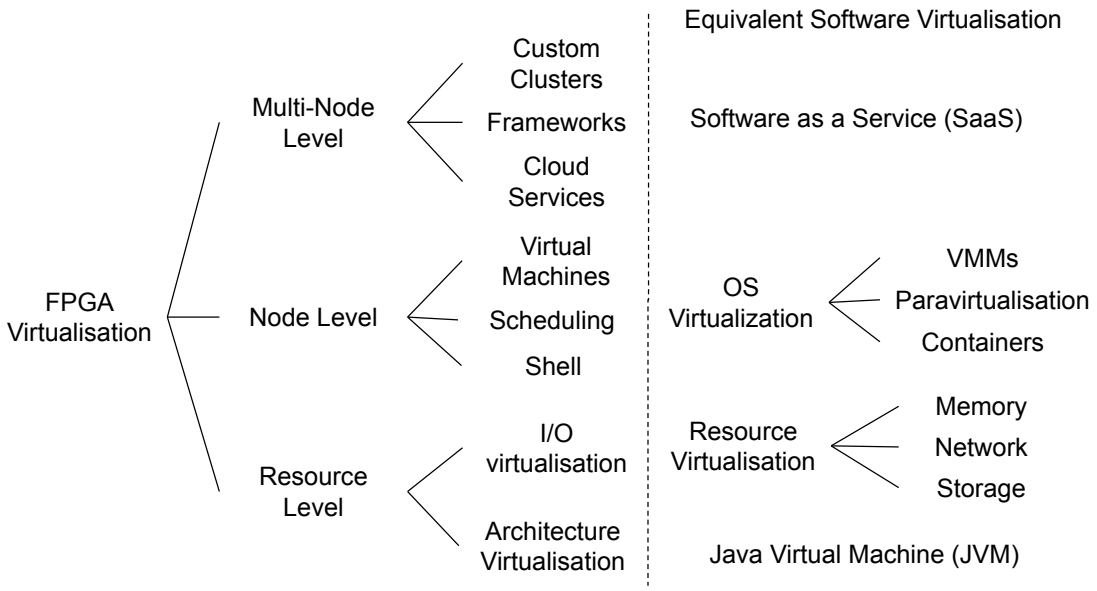


Figure 2.1: Classification of FPGA virtualisation techniques and their software equivalent.

These levels are defined as follows:

- *Resource level*: A resource on an FPGA can be of two types: reconfigurable or non-reconfigurable. Hence, for this level, we consider architecture virtualisation and I/O virtualisation. Examples at this level include Overlays [12, 21] for architecture abstraction and transparent I/O sharing in multi-tenant system [59, 117].

- *Node level*: We define a node as a single FPGA. Thus, we consider infrastructure and resource management techniques for this level. Examples at this level include VMM support [16, 115], runtime systems [5, 109, 126] and shells (also called hardware infrastructure of FPGA OS and Hypervisor-vFPGA approach) used to serve multiple concurrent user accelerators [13, 16, 30, 88, 139].

- *Mutli-node level*: We define multi-node as a cluster of multiple FPGAs. Hence for this level, we consider techniques and architectures used to connect multiple FPGAs for accelerating a job. Examples at this level include MIT's Leap [33], MapReduce [107, 116, 134] and Microsoft Catapult [88].

2.3 Virtualisation at Resource Level

In contrast to standard CPU/software virtualisation, FPGAs have to virtualise two distinct types of resources: reconfigurable and non-reconfigurable (i.e. I/O). These resources operate in a fundamentally different manner from each other. The reconfigurable resources are based on the FPGA architecture and represent the fabric onto which the reconfigurable accelerators are mapped. This mapping process varies from architecture to architecture. Consequently, the accelerators are required to be resynthesised when used across different systems, which can take anywhere from minutes to days, depending on the complexity of the design. Hence, the virtualisation at this level for reconfigurable resources aims at providing the portability of accelerators and rapid compilation by mapping accelerators to an abstract architecture and then using a tool or interpreter. Generic hardware descriptions also serve as a methodology to abstract from a specific target architecture and to provide portability across FPGA vendors.

Whereas, the non-reconfigurable resources represents the I/O resources, which are also found in the CPU/Software domain. Thus, these resources require similar abstraction and security measures as in software but with hardware support on FPGAs. This can include hard IPs such as embedded CPUs or memory controllers as well as soft-logic that is considered static at runtime. The virtualisation techniques employed for both types of resources are discussed in following subsections.

2.3.1 Overlays

Overlays provide another level of programmability that is implemented on top of the low-level FPGA resources, as shown in Figure 2.2. They are commonly used to improve productivity, allow runtime compilation, or to support portability of functionality across different systems (and even different FPGA vendors).

Overlays allow the compilation process to be split into two parts and decreases the compilation time required for generating an accelerator considerably when the CAD tool part can be omitted. Note, Java Virtual Machine's byte-code relates to native machine code as the overlay's application binary relates to the configuration of the overlay implemented on top the FPGA fabric. Similarly, like the byte-code translation to native machine code for achieving better performance, overlay applications may be directly translated into configurations of the underlying FPGA architecture [53, 63].

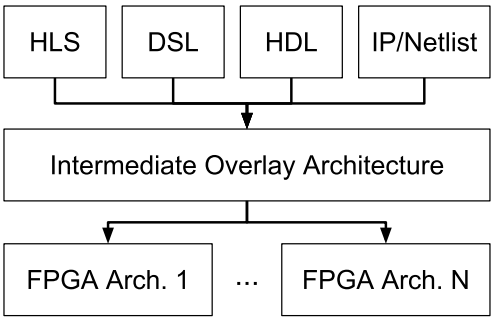


Figure 2.2: Overlay design flow.

Overlay architectures can range from multi-core systems to custom processing elements (PE) to custom LUT types, depending on the application or productivity requirements.

Jain provided a comprehensive overview of coarse-grained FPGA overlays in his PhD thesis [51] which are programmable at the data-word/operator level. Concisely, coarse-grain FPGA overlays can be soft-core processors, either from academia [15] or from industrial vendors [3, 122], vector processors [18, 93, 94, 135, 137], and connected arrays of processing elements (PEs) [21,23,36,65,96], in which programmable PEs and interconnects are provided. The motivation behind these soft-core processor approaches tends to be the provisioning of a more familiar programming environment for software developers, in contrast to the fine-tuned hardware accelerator implementations which target performance. In particular, vector processors based on multi-ALU parallelism have been shown to achieve a significant speed-up compared to soft-core counterparts. Examples include DySER [36], Venice [93] and Vegas [18].

The Firm-core project by Lysecky et al. [69] is an early example of a fine-grained overlay where the entire overlay programming was carried out by user logic implemented on top of the FPGA. ZUMA [12] improved implementation cost by mapping overlay LUTs and overlay multiplexers into LUT-configuration of the hosting FPGA fabric. Koch et al. [63] improved the implementation cost further by mapping the LUTs and the overlay routing directly onto the underlying FPGA routing fabric.

Overall, the extra level of programmability through an overlay comes at a substantial cost that needs to be justified. An example of this is the DRAGEN chip for DNA processing[1] where the overlay abstraction allows domain experts (that are not FPGA experts) to benefit from FPGAs. Another example where overlays can be better than

[1] https://www.illumina.com/products/by-type/informatics-products/dragen-bio-it-platform.html Accessed: 18 Feb 2020

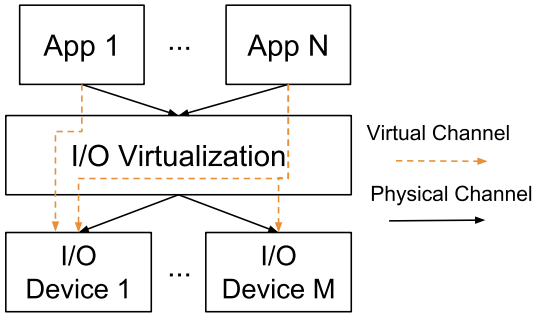


Figure 2.3: I/O virtualisation architecture, where an application can access multiple different I/O devices as if they were one or share the I/O devices transparently.

traditional HLS or RTL accelerators are situations that require rapidly changing functionality (at a speed that cannot be met using partial reconfiguration). For instance, VectorBox commercialises vector processor overlays where the same compute substrate is used for various concurrently running tasks under real-time constraints [94].

2.3.2 I/O Virtualisation

I/O virtualisation aims at managing I/O resources such that multiple applications can share the same resource or access multiple different resources with the same interface. Figure 2.3 shows an example where virtual channels are established between I/O devices and applications which do not correspond to the physical channels available for the I/O devices. The I/O virtualisation layer in the middle can be used to enforce the security measures (e.g., in memory virtualisation [13, 59]), hide complexity of the I/O interface [59], monitor resource usage and enforce QoS (e.g., in cloud systems) [16], as well as optimising access time (e.g., provide buffers for memory load and stores).

Fundamentally, the virtualisation support for I/O is the same as in CPU/software, with the main difference being the implementation technology. For FPGAs, the virtualisation can be either implemented in a software domain (e.g., by using a soft-core [13] or a host CPU [30, 103]) or a dedicated hardware module [59, 117, 139] using some reconfigurable resources. The software approach tends to be used for high flexibility or to save more reconfigurable resources for application logic. Whereas, the hardware approach tends to be used for high performance I/O access and management.

Moreover, the I/O virtualisation layer can also be used to assist a CPU for higher performance I/O access rather than just serving the accelerators on the chip. For example, Abbani et al. [1] have shown how FPGAs can be used to accelerate storage up to 6× performance for data-intensive applications running on a distributed reconfigurable

active SSD platform. While Chalamalasetti et al. [14] and Lavasani et al. [66] showed an FPGA accelerator-based solutions for Memcached. Further, Microsoft uses FPGAs in the Catapult project to reduce network traffic to CPU by delegating the majority of NIC requests directly to FPGAs [88]. Note, these are just a few examples which are applied to accelerate I/O by offloading compute-intensive work to an FPGA as a middleware.

2.4 Virtualisation at Node level

Virtualisation support at the node level represents the infrastructure (in both hardware and software) required to manage the resources related to a single FPGA. It is useful to split the node level virtualisation into three different categories: 1) Virtual Machines Monitors (VMMs), 2) shells, and 3) scheduling, as described in the following subsections.

2.4.1 Virtual Machine Monitors

The Virtual Machines Monitors (VMMs) are a standard way of performing virtualisation in the CPU world and tackle various challenges which apply to FPGA virtualisation. Thus, it is only natural to extend VMMs to support FPGAs. Wang et al [115] made one such attempt, where an FPGA accelerator integration took place at the lower device driver level in the Xen VMM. Their experiments show that VMMs implementation can access FPGA accelerators with close-to-zero overhead compared to without VMM layer and at the same time share the accelerator among multiple operating systems. This approach is a good fit for the applications which require only *static acceleration support* and tight coupling of VMMs on a host CPU with an FPGA accelerator. The micro-kernel approach by Xia et al. [120] takes this approach one step further and provides multiple partial regions and resource sharing mechanisms for VMMs while Jain et al. [52] use a micro-kernel with overlay virtualisation to also provide accelerator portability. Chen et al. [16] presents another similar effort for integrating Xilinx FPGAs into Linux-KVM with OpenStack support.

Overall, all these prototypes provide isolation between multiple processes or multiple virtual machines, with some support for resource allocation and they aim at fulfilling the following virtualisation objectives: Multi-tenancy, Resource Management, Isolation, Security and Resilience.

Moreover, FPGA virtualisation with VMMs integration tends to adopt the conventional model of treating CPU as a first-class citizen and FPGAs as a special peripheral. This makes the incorporation of FPGAs in software domains a little easier for the software programmers since accessing an FPGA accelerator tends to become a simple library call with a similar interface as a GPU. Hence, applications can scale to use multiple FPGAs by employing standard software frameworks used for GPUs, with minor modifications.

2.4.2 Shells

Shell is a recent term used to describe the static part of the FPGA system and is often referred to as *Hardware infrastructure of FPGA OS* or *Hypervisor for vFPGA*s [59]. In the past, FPGA operating systems such as ReconOS [67] and Borph [99], which allowed the hardware accelerators to access software operating system services via communication and synchronisation primitives of threads or UNIX process interfaces, aimed at making hardware accelerators an equal counterpart to the software processes. However, the current virtualisation needs are different and aim at providing hardware acceleration services to the software processes. Hence, these shells are designed to be lightweight and only provide the common infrastructure required for different applications, i.e. all of the I/O virtualisation support, resource management and the drivers required to program the hardware accelerators.

There are many ways in which a shell can provide these requirements. Figure 2.4 shows some of the infrastructures proposed in the recent works which target virtualisation in particular. Table 2.1 lists various approaches for shells where a platform provides a separation of application logic from the OS logic based on the execution models they support.

At this level of virtualisation, the execution models are the prime factors for deciding which functionality is required to be a part of shells, and these can be categorised into four types:

- *SFSA*: Single FPGA Single Application
- *SFMA*: Single FPGA Multiple Applications
- *MFSA*: Multiple FPGAs Single Application
- *MFMA*: Multiple FPGAs Multiple Applications

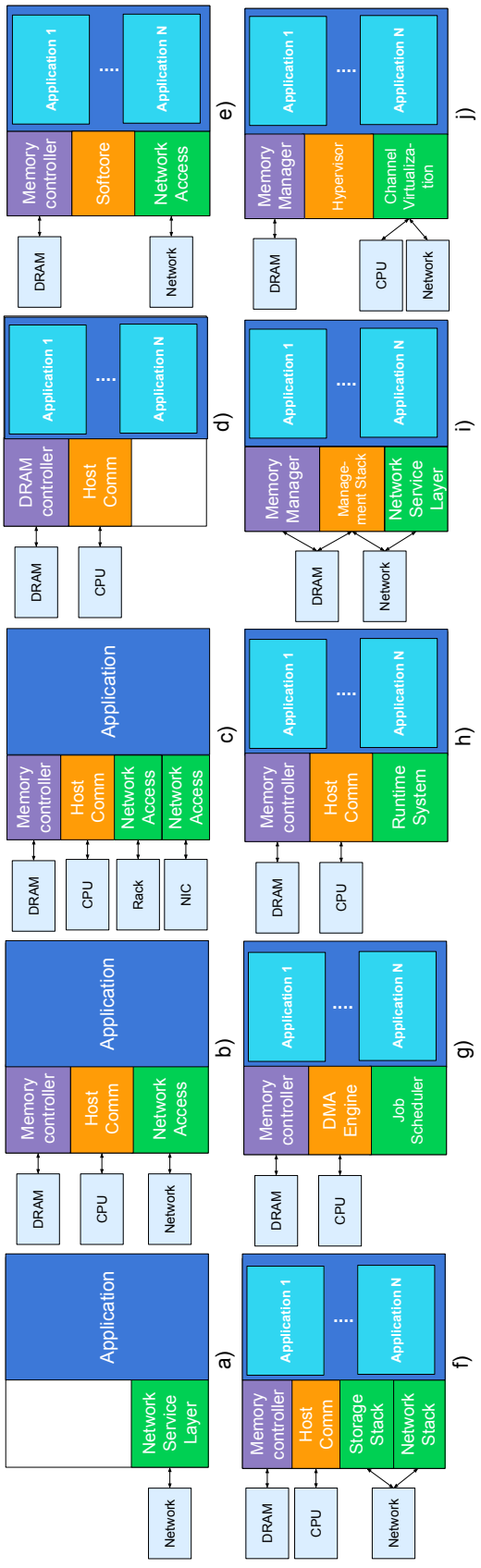


Figure 2.4: Proposed FPGA infrastructure and shells for hosting one for more reconfigurable applications with virtualisation where a) is Network-attached [118], b) is Tarafdar et al. [103], c) is Catapult [88], d) is Fahmy et al. [30], e) is Byma et al. [13], f) is Feniks [139], g) is Chen et al. [16], h) Asiatici et al. [5], i) IBM Zurich [117], and j) Knodel et al. [59].

In most cases, multiple applications on a single FPGA are achieved using multiple partial regions. These regions can be of identical shapes as well as different shapes. The different shaped approach is usually taken to support multiple different sizes of modules without having to reconfigure the entire FPGA [140](by loading the accelerator to the required size region) using standard partial reconfiguration (PR) flow [124]. Whereas, identically sized partial regions (also called tiled regions or resource slots) are used to combine partial regions to host bigger accelerators with customised PR flow. The best example of this is the Erlangen Slot Machine [10] which was designed with flexibility in mind. In a slotted shell, an application can occupy one or more adjacent slots, providing more flexibility for the resource allocation and reduces internal fragmentation [60].

Moreover, each execution model requires some form of connectivity and this can be categorised into three different classes: i) host connectivity, ii) independent connectivity, and iii) hybrid connectivity. Shells which support only host connectivity let a host CPU handle the resource management of the FPGA to a certain extent and use the majority of reconfigurable resources for the applications [30]. With the availability of embedded CPU cores providing a rich set of peripherals (e.g., as available in ZYNQ Ultrascale+ FPGAs), this model is now feasible as a single chip solution. However, for datacenter/cloud environments, it can lead to under-utilisation of resources as the FPGA allocation is tightly coupled with the CPU allocation. Thus, there are proposals which recommend independent connectivity for these scenarios, which allow sharing of FPGA resource among multiple different CPUs [117, 118] or even act as a standalone device for the application [13, 88, 102, 117]. This flexibility often requires FPGA support for the network layer and other I/O resources (e.g. Ethernet and memory controllers), which can occupy a considerable amount of the available FPGA resources in some cases (as shown in Table 2.2). In contrast to host connectivity and independent connectivity, the hybrid approach aims at supporting both forms of connectivity to benefit from i) offloading control intensive or complex resource management tasks to CPUs but also ii) use the dedicated hardware on an FPGA to accelerate I/O accesses if required (as done in Microsoft's Catapult platform [88]).

2.4.3 Scheduling

Scheduling, in general, is a well-established topic in software systems, specifically for multi-tenancy and for improving utilisation of resources. The conventional techniques of scheduling are preemptive, non-preemptive, and co-operative scheduling which can

Table 2.1: Hardware platforms and their application support. Note that the certain platforms may be able to scale multiple FPGAs via CPU, but without the mention of network support for scaling in the paper, we refrained from including them into the multi-FPGA category.

		Applications	
		Single	Multiple
System	Single FPGA	[30], [114], [103], [88] [119], [140], [47], [118] [113], [50], [27], [97] [104], [138]	[13], [59], [139], [16] [10], [112], [119], [5] [140], [47], [30], [138]
	Multi-FPGA	[118], [103], [88], [113] [27], [104], [138]	[13], [139], [117], [59], [138]

be used to share the FPGA in the *time domain*.

However, these techniques cannot be applied directly for all types of FPGA accelerators as the state space which require saving for an FPGA accelerator is, in general, non-trivial if we perform a context switch at an arbitrary time. This is because the state may be spread out across Flip-Flops, Logic Cells and BRAMs. Saving and restoring the complete state for such a case can easily take milliseconds [39, 58] in addition to partial reconfiguration latency. Further, the hardware support required for preemptable hardware module tends to be restrictive and very target specific (see Section 3.4.2 for details).

Rupnow et al. [89] proposes a non-preemptive policy that does not induce this without hardware restrictions, where a hardware task is either blocked, dropped or rolled back to CPU if a context switch needs to be performed. The non-preemptive approach simplifies the design and can be implemented at relatively low-cost because the accelerators run to completion. In contrast to both the preemptive and the non-preemptive approach, co-operative scheduling employs context switching when an application reaches an execution checkpoint for minimal overhead [7, 43, 109, 120].

The trade-offs involved in different types of hardware context-switching methods will be discussed further in Section 3.4.2.

Table 2.2: Comparison of state-of-the-art infrastructure present at the node level. For scalability we consider the support of connectivity in the shell (Low: host connection only, Medium: network connection, & High: both host and network connection), for flexibility support for range of acceleration models is considered, i.e. custom accelerators, framework, and bump in the wire (Low: 1 model, Medium: 2 model, & High: all 3 models), for programmer productivity we consider if shell supports RTL, HLS/DSL and frameworks (Low: only RTL, Medium: HLS / DSL + RTL, & High: HLS/DSL + RTL + frameworks), for isolation we consider how advance support is provided by component managers (Low: 1, Medium: 2, & High: >2 resource multiplexing) and finally for utilisation we consider the number of regions and scheduling capabilities (Low: 1 region, Medium: >1 region, & High: >1 region + spatial scheduling). *The only reported area metric.

OS infrastructure	FPGA Area	Multi-tenancy	Scalability	Flexibility	Programmer Productivity	Isolation	Utilisation
Byma et al. [13]	74% (BRAM)	✓	Medium	Medium	Medium	Low	Medium
Chen et al. [16]	41% (Logic)*	✓	Medium	Medium	High	Medium	Medium
IBM Zurich [117]	33% (Logic)*	✓	Medium	Medium	Medium	High	Medium
Fahmy et al. [114]	7% (Total)	✓	Low	Medium	Medium	-	Low
Catapult [88]	76% (Total)	✗	High	High	Low	-	Low
Network-attached [118]	32% (BRAM)	✗	Medium	Medium	Medium	-	Low
Tarafdar et al. [103]	19.56% (BRAM)	✗	High	Medium	High	-	Low
Feniks [139]	13% (Logic)	✓	High	High	High	High	Medium
Knodel et al. [59]	42% (Logic)	✓	High	High	Medium	High	Medium
Asiatici et al. [5]	-	✓	Low	Low	Medium	Medium	High
ArmophOS [57]	-	✓	High	High	Medium	Low	High
Amazon F1 [92]	34%	✗	Low	Medium	Medium	Low	Low
ECOSCALE [74]	43%	✓	High	High	High	High	High
ZUCL 2.0 [85]	20%	✓	Low	Medium	Medium	Medium	High

Spatial Domain Scheduling for FPGAs

Many related approaches have hypothesised and theoretically investigated placement and scheduling approaches in the *spatial domain* to avoid fragmentation and for improving FPGA utilisation [4, 19, 25, 26]. In these publications, theoretical and simulation results show that scheduling in the spatial domain can potentially improve performance for FPGAs. However, only very recent research is investigating possible implementations of FPGA virtualisation in the spatial domain. Asiatici et al. [5] proposed dynamic scheduling which can take advantage of the free slots available at runtime to maximise the utilisation and thus the performance. This scheduling approach maximises the number of pipeline instances when a task is created and keeps this allocation untouched until task completion. A dynamic scheduling technique which can potentially increase or decrease accelerator's resource consumption according to the workload requirements at runtime is a particularly desirable feature when moving forward in multi-tenant systems.

Scheduling for CPU+FPGA Systems

Many researchers have proposed various ways to describe the computation for multiple device types (i.e. CPU, GPU, FPGA, and ASICs), such as the thread-based model of ReconOS [67] and Hthreads [2, 81], the OpenCL programming model [77], and domain-specific languages (DSL) (e.g., HeteroCL [64]). A programmer either has to select the device type by instrumenting the code [67, 73] or passing hints/profiling information to the runtime as meta-data [7, 20]. These unifying task descriptions then allow a centralised scheduler (or runtime system) to allocate resources on different devices with guarantees to achieve the same result.

For proposals with hardware thread implementation [67, 81], a standard software OS scheduler performs the scheduling, and once the device type is selected, the task runs to completion [2], hence, achieving task partitioning between different devices. Beisel et al. [7] proposed an extension of this, by using cooperative scheduling to allow multi-tasking on heterogeneous platforms with *time sharing*. However, despite the flexibility to reallocate resources between different device types, that work assumed the FPGA accelerators to be static, i.e. did not leverage the reconfigurability of FPGAs.

Table 2.3: Comparison of different optimisation parameters considered by different CPU+FPGA schedulers. Where multi-tasking is the ability to share resources between multiple different tasks/users, device type selection is automatically choosing the appropriate device for execution, work-load partitioning is the sharing of work across multiple devices, number of compute units is ability to change the number of execution units, and accelerator type selection is the ability to choose implementation of accelerator(execution unit) on same device.

Past Work	Multi-tasking/ Task partitioning	Device type selection	Work-load partitioning	Number of compute units	Accelerator type selection
ReconOS [67]	✓	✓	✗	✗	✗
BORPH [99]	✓	✓	✗	✗	✗
Hthreads [2]	✓	✓	✗	✗	✗
Beisel et al. [7]	✓	✓	✗	✗	✗
Asiatici et al. [5]	✓	✗	✗	✓	✗
ArmophOS [57]	✓	✗	✗	✗	✓
Cong et al. [20]	✗	✓	✓	✗	✗
Nunez-Yanez et al. [79]	✗	✓	✓	✗	✗
Guzmán et al. [37]	✗	✓	✓	✗	✗
Huang et al. [45]	✗	✓	✓	✗	✗
Our proposal in Section 4.3	✓	✓	✓	✓	✓

The runtime systems for OpenCL and DSL, in contrast to hardware threads, often *focus on a single application* and consider optimisation such as workload partitioning between different devices (i.e. distributing data-parallel work across different devices). This collaborative execution is based on either static partitioning factors [37, 45] or identified at runtime by profiling [20] or dynamic scheduling of data-parallel work [37,45,79]. These methods have been shown to improve performance by exploiting idle units available in a system. Overall, with the rise in the heterogeneous platforms providing both CPU and FPGA[2] these approaches can improve performance and energy efficiency. However, at present only device type selection and workload partitioning are considered as optimisation parameters (see Table 2.3). In particular, with the reconfigurability, FPGAs allow selecting the accelerators type and the number of compute units at runtime. Optimising these additional parameters combined with support for multi-tasking is an open research problem, solving which can help unlock better performance and energy efficiency for heterogeneous systems.

[2]In some cases, even GPUs [125].

Scheduling for Clusters

Specific scheduling approaches have been proposed to target a FPGA as a Service (FaaS) and an Acceleration as a Service model (AaaS). One such approach is mapping multiple users who require the same accelerator functionality on the same FPGA such that partial reconfiguration is not required to serve multiple users [44, 48]. Another approach has been implemented for VineTalk by Mavridis et al. [73], to provide support for sharing an FPGA within a native server, a virtual machine, or a container in a heterogeneous datacenter. The user can select the appropriate accelerator type (e.g., GPU or FPGA) based on workload or algorithm specifics through this API. With the adoption of OpenCL as a de-facto standard for heterogeneous computing and proposals like SparkCL [91] for bridging the gap between Java and OpenCL, these techniques seem to be leading the way for the heterogeneous computing paradigm at the cluster level.

2.5 Virtualisation at Multi-Node level

The aim of virtualisation at multi-node level is to map an acceleration job across multiple FPGAs in a transparent manner. To provide this abstraction, a virtualisation model must provide communication among multiple FPGAs and abstract the details of how they connect from the user. Applications at this level often run on a data-parallel piece of the problem where each FPGA computes the whole job or where multiple FPGAs accelerate a problem in conjugation with each other.

Multiple FPGAs can commonly be connected in three different ways, as shown in Figure 2.5. These are i) FPGA-to-FPGA architectures where FPGAs directly communicate with other FPGAs, ii) a server-client architectures where a server is a remote CPU issuing workload to the independent FPGAs, and iii) the traditional server-client architectures where both server and client are CPUs, and FPGA is a special hardware peripheral attached to a client CPU. Note that systems may combine these models to form a *hybrid architecture* based on the application requirements. For instance, Microsoft's Catapult [88] uses an architecture with directly linked FPGA (as in a) of Figure 2.5) that are connected to a CPU node as shown in variant c) in Figure 2.5.

These architectures are the underlying base for the three virtualisation models as discussed below: custom clusters, frameworks, and cloud services.

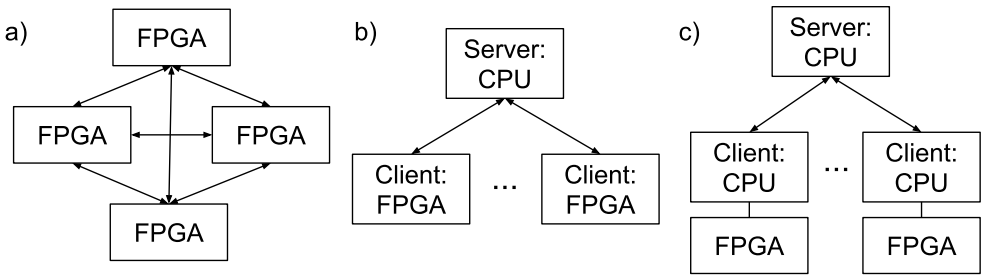


Figure 2.5: Architectures used to scale an acceleration job across multiple FPGAs. Where a) represents FPGA to FPGA communication, b) shows a server-client architecture where FPGA directly communicate with the server and c) denotes the server-client architecture where FPGAs are special peripherals to the client CPU.

2.5.1 Custom Clusters

In the custom cluster approach, the computation is split among multiple FPGAs, where data is passed after processing from one FPGA to another by FIFO/file/network semantics. The approach often follows a systolic arrays model where each Processing Elements (PE) is an FPGA. Examples of this FPGA-to-FPGA architecture are Leap [33], ViTAL [138], and Maxeler MPC-C and MPC-X series [82]. Leap requires the user to annotate the code with special pragmas to create FIFOs which are latency insensitive to communicate with a different FPGAs. These FIFOs send data using high-speed serial transceivers and expose the content again in FIFO buffers on the other side of the channel. A similar approach is taken by Maxeler when connecting multiple FPGAs using a proprietary point-to-point connection protocol called MaxRing [82]. As an alternative, an automated version of this approach is presented in ViTAL [138] with partial reconfiguration support.

Accelerators can also be designed to communicate directly with other nodes by explicitly using the network connections, such that data movements and compute is tailored to the target application. An example in this category is Levenshtein's distance implementation for CUBE (a one-dimensional cluster of 512 FPGAs) which exploits a systolic architecture for stream processing across multiple FPGAs [136].

2.5.2 Frameworks

The framework model for virtualisation takes a similar path as the software world and adopts the server-client architecture for deploying accelerators. A server CPU is responsible for configuring the FPGAs and managing application data, while FPGAs are

responsible for performing the actual computation. Various data management techniques have been studied before extensively for distributed systems using CPUs, and the same techniques can potentially be reused for FPGAs. In particular, frameworks like Map-Reduce can be supported with FPGAs by implementing the map and reduce functions as FPGA accelerators, where the data is distributed to FPGAs and collected back after the computation in the standard Map-Reduce manner [95, 116, 134].

Furthermore, frameworks can also help to bridge the heterogeneity between devices, as the programmer writes the application against a fixed interface. An example of this is Axel, where Map-Reduce is used to form heterogeneous clusters providing FPGAs and GPUs [107]. However, to implement this, a JVM based framework needs to be extended to support FPGAs efficiently. Chen et al. [17] has shown that JVM-to-FPGA communication overhead can be significant and it requires careful handling for frameworks such as Apache Spark to minimise communication and data movements.

Moreover, the support for an industry-standard language for heterogeneous computing such as OpenCL has been investigated. To virtualise FPGAs using the SDAccel framework for OpenCL from the vendor Xilinx, Tarafdar et al. [103] propose using an MMU layer to map data across multiple FPGAs (i.e. allocating data across multiple FPGAs and using a directory to fetch/send the data transparently from/to remote nodes). Further, Iordache et al. [48] proposed the concept of FPGA Groups to share one or more physical FPGAs configured with the same accelerator. Here the granularity of resource allocation is a whole FPGA which may not lead to high utilisation. However, an auto-scaling algorithm allows to grow or shrink the number of FPGAs in a group dynamically. Huang et al. [44] proposes a similar model for their Blaze runtime, which aims at implementing FPGA as a Services (FaaS). Blaze not only extends Hadoop YARN (a cluster management system) with accelerator sharing among multiple computing tasks but also reduces the required programming effort considerably for systems like Apache Spark and YARN.

2.5.3 Cloud Services

The cloud service model completely abstracts the details of where the computation is taking place from a user. A user is only guaranteed quality-of-service (QoS) and correctness of the output, and thus in the background, an FPGA can be employed to perform computing instead of a CPU. Note, this approach is different from providing FPGA as a Service (e.g., EC2 service by Amazon [92]), as it does not relate to the provisioning of FPGAs but provisioning of the application/web-service itself. Microsoft

took this approach for accelerating the Bing web search ranking algorithm [88] to achieve 95% higher performance at the cost of 10% higher power consumption. Baidu has reported similar results for accelerating Deep Neural Networks (DNN) [80]. Moreover, the runtime system can utilise FPGA as co-processors to accelerate compute-intensive kernels in a high-performance computing environment. An example of this is the work done by El-Araby et al. for Cray XD1 [28] using FPGAs as co-processors for the Single Program Multiple Data paradigm.

2.5.4 Hybrid Architectures

There are also hybrid architectures which can support multiple forms of connectivity based on the FPGA infrastructure available at the node level. An exemplar of hybrid architecture is Microsoft's Catapult project [88] which allows the acceleration of tasks as a special peripheral connected to a host CPU core but can also communicate among FPGAs directly in a standalone fashion. FPGAs with network access can support all connectivity architecture as a CPU can be provided as a soft-core or an embedded SoC on an FPGA. The most common technique to manage these FPGAs is to use OpenStack for provisioning of the FPGAs and letting the user connect and program the FPGA using an IP or MAC address of the FPGA or vFPGA [13, 103, 118]. This gives the user the flexibility to connect to the FPGA using remote procedure calls or socket connections. Moreover, this complexity can be further abstracted away by using frameworks and libraries for typical applications [73].

2.6 Discussion

There are multiple directions and levels at which virtualisation is required and employed, as mentioned in previous sections. However, these virtualisation approaches are not entirely isolated from each other. Figure 2.6 shows how these individual developments can be combined for a complete FPGA solution at various execution levels. For example, I/O virtualisation can be implemented as part of the standard system functionality and Overlays can be compiled for the PR region to lower reconfiguration latency, improve portability and ease of use for non-FPGA experts. Virtual machine monitors and containers can then mediate and isolate the users in a software environment while providing access to FPGA shell services. The scheduler for the VMM or other middleware can dynamically manage the FPGA resource allocation in the time

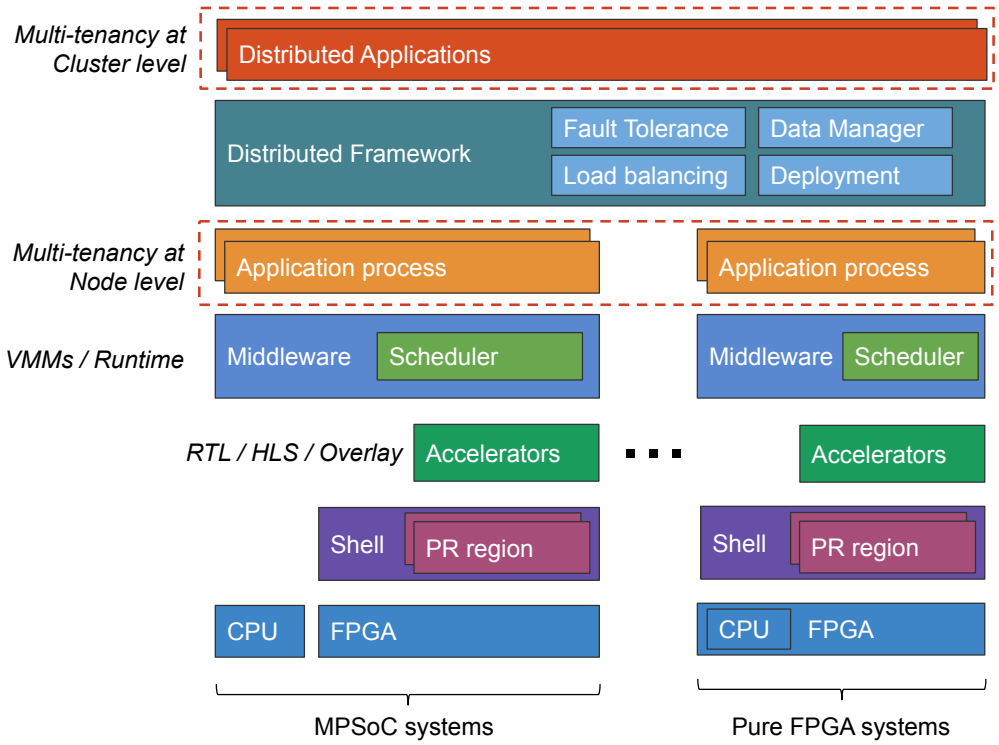


Figure 2.6: A complete FPGA acceleration stack with various levels of virtualisation.

domain and transparently from the user. Distributed frameworks can be used to share the application workload on a custom cluster where each node is providing multi-tenancy support for efficiency and high-performance computing.

However, despite these advances in FPGA virtualisation at various levels, the current FPGA ecosystem is suffering from three major issues:

- *Lack of modularity and portability in system infrastructures*: The current FPGA development flow is rigid and assumes many direct dependencies between system components (for both hardware and software components), which do not allow them to be replaced or ported to another system without significant effort or recompilation (see Section 3.3). This severely affects the developer productivity and maintainability of the systems in the long run. Existing work on FPGA operating systems [67, 99] achieved decoupling between hardware and software with heavy interface wrappers (threads and UNIX processes). However, the new needs of virtualisation require more light-weight solutions (e.g., at library function call level) and further decoupling among hardware components (shell and accelerators) as well as software components (libraries, runtime system and drivers).

- *Lack of adaptability to changing workloads*: Runtime resource allocation is typically tied to the time-domain only, ignoring the spatial dimension for which the FPGAs are often deployed. This leads to under-utilisation of resources in multi-tenant systems and makes the overhead of context-switching more pronounced. Moreover, there is a lack of heterogeneous systems which can automatically move computation between different types of devices, such as CPU and FPGA, for performance optimisation while supporting multi-tenancy. Hence, these heterogeneous system often cannot realise the potential performance achievable by using various types of accelerators, devices, or number of compute units for application execution.

- *Lack of adaptability in FPGA clusters*: Since FPGA accelerators often execute with a run-to-completion model, most systems refrain from dynamically moving an accelerator from one FPGA to another during execution for load-balancing, maintenance, or fault-tolerance purposes. Given that, at the cluster level, these are the most common operations and are required to provide resilience and quality-of-service guarantees, live migration for FPGA accelerators is an essential feature. However, the existing solutions for accelerator migration employ preemptive context-switching and blocking migration mechanisms (see Section 3.6) which lead to restrictions on accelerator design and expensive penalty [58]. A new solution which can transparently move an accelerator from one FPGA to another with low overhead is required.

Furthermore, solutions to these issues must 1) provide ease of use for both hardware and software developers, and 2) be able to support standard hardware accelerator design practices, i.e. i) avoid addition of external logic to reduce performance and resource overheads, and ii) remain applicable to existing accelerators as much as possible to allow widespread adoption in industry and research environments rather than being application-specific solutions.

Hence, in this thesis, we tackle these three core issues by 1) building a modular FPGA OS with support for 2) dynamic reallocation of heterogeneous resources and 3) low overhead live migration to enable large scale development and deployment of FPGA systems with higher efficiency and performance while using existing applications and tools. Following Chapter 3 and Chapter 4 describes and discussed these solutions and their implementation in further detail, respectively. The final system evaluation for various use-cases and workloads can be found in Chapter 5.

Chapter 3

Modularity and Resource Elasticity: Concepts, Methods and Trade-offs

3.1 Chapter Overview

In this chapter, we introduce the concepts, methods, and trade-offs required to resolve the three key issues highlighted in Section 2.6 (lack of modularity, adaptability at node level and multi-node level) for the FPGA virtualisation ecosystem.

To resolve the first issue of lack of modularity and portability, we need an FPGA operating system which spans across both hardware and software levels. Hence, in Section 3.2, we describe the layered architecture of an FPGA operating system as well as the required functionalities and components at both hardware and software levels. We then discuss the sources of dependencies and how they can be mitigated by introducing a set of abstraction layers between development stages to achieve a modular FPGA operating system in Section 3.3.

To solve the second issue of lack of adaptability to dynamic workloads on FPGAs, we introduce the concept of 'resource elasticity' to allow FPGA accelerators to change their resource allocation in the spatial domain at runtime in addition to time-domain multiplexing in Section 3.4.1. The challenges and methods for performing a hardware context-switch on FPGAs to enable such a dynamic resource allocation are discussed in Section 3.4.2. With the ability to allocate resources in the spatial domain at runtime, the resource elastic schedulers face a new set of trade-offs compared to traditional time domain only schedulers and these will be discussed in Section 3.4.3·

Moreover, modern FPGA systems often include an on-chip CPU; hence, to maximise the system utilisation, it is essential to schedule execution on both CPU and

FPGA together. However, this requires to consider additional optimisation parameters at runtime. Hence, Section 3.5 describes the standard system infrastructure of modern FPGA systems and corresponding optimisation parameters in detail. To enable the dynamic resource allocation across these different types of devices (CPU and FPGA), we need to be able to migrate computation between them. Section 3.5.1 will discuss the techniques to perform this while Section 3.5.2 describes the common execution model (OpenCL) to support different heterogeneous devices with the same source code.

To address the third issue of lack of adaptability in multi-node FPGA systems, we need a low overhead migration scheme which can hide the latency of transferring data and compute to another node from the user and allow the cluster manager to adapt resource allocation for performance optimisation and reliability. Hence, in Section 3.6, we propose and describe the methods for live migration of OpenCL FPGA accelerators which fulfil this requirement.

Finally, in Section 3.7 we summarise and detail the system requirements to build a modular FPGA operating system which can support dynamic workloads and solve the three key issues. We will discuss the implementation details of the concepts and methods adopted for this operating system in Chapter 4.

3.2 FPGA Operating System

FPGA accelerators can provide high-performance computing at very low energy cost for applications ranging from neural-networks to multi-media acceleration to network processing, and this has brought FPGAs to large-scale deployments as datacenter devices in the cloud and as embedded systems at the edge. However, to sustain this broad scope of requirements present in terms of heterogeneity of devices, operation environments, EDA tools, and users and developer needs, we require a standardised way to manage and integrate FPGA system components, in other words: we need an FPGA operating system.

Unlike standard CPU operating systems which consist only of a software infrastructure, an FPGA operating system requires an infrastructure at both the hardware and the software level (as shown in Figure 3.1). The following subsections describe the role of each level in detail.

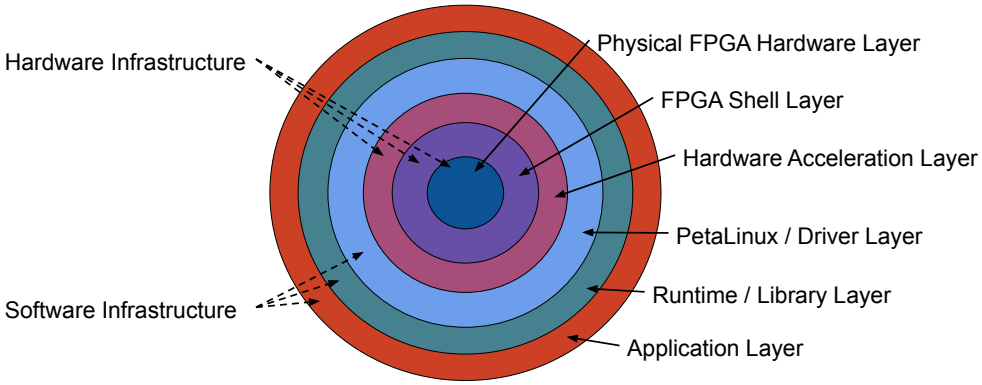


Figure 3.1: The layered architecture of an FPGA operating system.

3.2.1 Hardware Infrastructure

The hardware infrastructure is implemented directly on top of the physical FPGA resources and is commonly built using a partial reconfiguration (PR) flow (e.g., Amazon F1 FPGA instances [92]). A PR flow generates a system in two partitions: a static system and one or more reconfigurable regions. An FPGA shell is essentially the static system of the PR flow and can be considered hardware equivalent to a software OS kernel. It provides common functionalities required by the accelerators such as an interconnect, network, and memory access as well as (optional) other I/O and management IPs necessary for the target system. The counterpart of the shell is the partially reconfigurable region (also called a slot) which can host different hardware accelerators at runtime. A shell which supports multiple partial regions can support multi-tenancy in the spatial domain by allowing multiple accelerators to execute in parallel. Figure 3.2 shows an example of such a shell. However, the number of partial regions, their location and sizes depend on the system requirements and changes from system to system. Hence, an integral part of the shell is the compilation flow required to map the hardware modules onto the resources of a PR region with the required physical interface for the target system infrastructure.

Moreover, a shell often includes a CPU, either in the form of a soft-core (e.g., on the datacenter FPGAs [126]) or a hardened CPU-core on MPSoC platforms used for embedded systems [126]. This CPU is used to host the software infrastructure which is necessary for managing the FPGA resources (see Section 3.2.2 for details).

With this, a shell provides the basic OS functionality at the hardware level to host hardware applications (accelerators) on an FPGA.

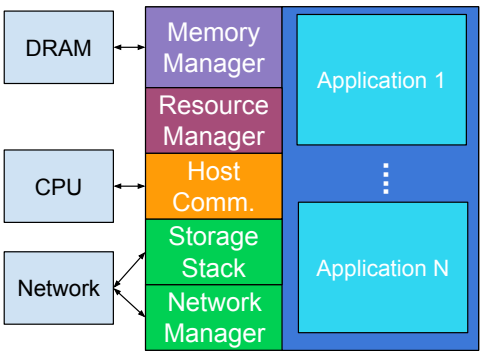


Figure 3.2: The overall organisation of an FPGA shell.

3.2.2 Software Infrastructure

The software infrastructure of an FPGA operating system serves as a middle layer between the hardware accelerators and the user applications (software) while executing on a host CPU (soft or hardened). However, unlike software operating systems, the software infrastructure for an FPGA OS should be resilient to dynamic changes in the underlying hardware and be portable to different FPGA boards while hiding the heterogeneity from the software programmer. Consequently, an FPGA OS is responsible for managing the heterogeneity of the hardware resources available on the FPGA as well as to provide the high-level APIs and software integration for ease of development. This requires the software infrastructure to provide five important functionalities:

1. A boot-loader and a kernel to bring up the FPGA system in an operational state.
2. A set of drivers and hardware abstraction layer (HAL) to communicate with the hardware accelerators.
3. A scheduler to dynamically allocate FPGA resources (i.e. partial regions) and hardware accelerators to software applications.
4. High-level standard libraries to make hardware acceleration easily accessible.
5. Integration with existing CPU software stacks to benefit from legacy code.

3.3 Modular Development Flow

To support the FPGA virtualisation requirements, FPGA operating systems need to provide better portability with low overhead, i.e. the introduction of intermediate layers and communication cost must be low in terms of the latency and the reconfigurable

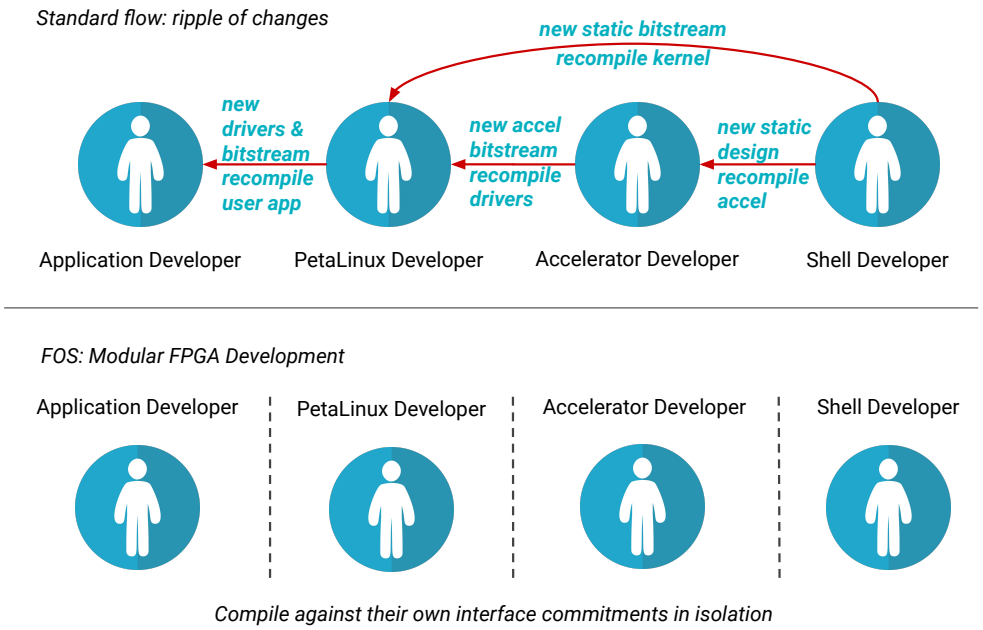


Figure 3.3: On an update of the shell IP or EDA tool version, the rest of the system components must be recompiled because of the dependencies in the standard tool flow (see Section 3.3). Ideally, each component should be able to compile in isolation with given interfaces, as provided in FOS.

resources required. However, to achieve this is difficult because hardware accelerators and boards require unique optimisations and implementations, leading to heterogeneity in FPGA shells, accelerators, EDA tools, and low-level software required to interface with it. This reduces not only the portability of a system but also its *maintainability*, which is a crucial aspect for any operating system (OS). A simple change in a tool version or the addition of a system IP can lead to a recompilation of the whole FPGA stack when using the current industry tool flow [56, 119], as shown in Figure 3.3. Essentially, implying that update in the operating system means recompilation of all applications which run on top of it.

The following subsections explain the causes of these dependencies during FPGA development flow and how we can mitigate them with low overhead.

3.3.1 FPGA Development Stages

There are five primary stages of development required for a multi-tenant/cloud FPGA system, as shown in Figure 3.4. The bottom two stages of this stack are hardware

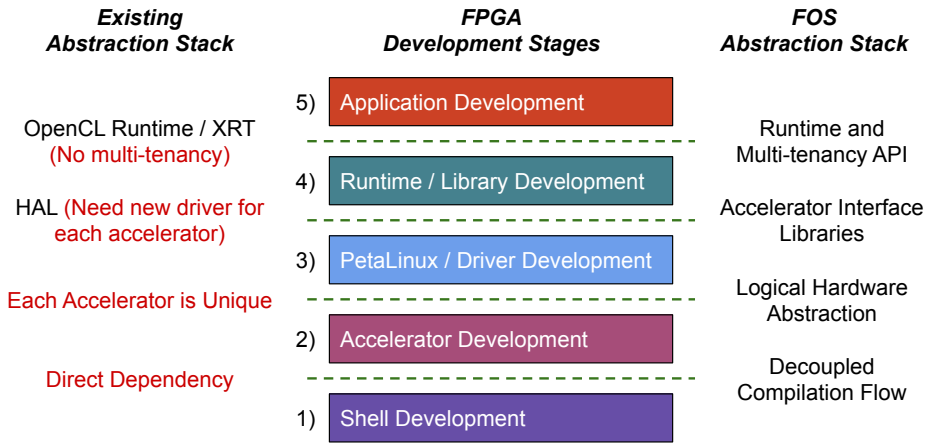


Figure 3.4: The FPGA development stages and the abstraction layers required between them to support modularity.

development stages which drive the application performance and the support for multi-tenancy (via PR).

In particular, 1) *shell development* requires designing and implementing the common system functionalities required by the user accelerators. An essential step at this stage is floorplanning which includes deciding the amount of resources to be allocated to accelerators and a definition of the interface exposed to the accelerators. This stage also requires identifying the address mappings at which the driver layer will access the accelerators.

While the shell development provides the infrastructure, its counterpart, 2) *accelerator development*, involves designing RTL or HLS accelerators with application-specific optimisations (commonly for the highest performance achievable with the allocated resource budget). To compile the accelerators for the shell, the compiler must know the exact resources available in the partial regions and the physical locations of the interface pins. Without this information, we cannot implement partially reconfigurable accelerators. However, with the current FPGA vendor partial reconfiguration development flows, accelerator compilation directly depends on the shell and requires knowing the implementation (internal resource allocation and routing) of a shell [129]. This is because accelerator modules are implemented as an increment to a specific shell. Hence, any update made to the shell requires recompiling of the accelerators. Note that this flow is used to build the most state-of-the-art shells [30, 57, 59, 117, 140].

The remaining three stages of the stack are software development stages which aim at programming the accelerators, performing runtime optimisations, and improving the

developer productivity.

The 3) *PetaLinux/Driver development* involves building the embedded software required to boot up the board with the necessary I/O and high-level functionalities as well as means to communicate with the accelerators in order to send and receive data. The current Xilinx tool flow expects the user to write a new driver for each accelerator or generate one automatically when using Vivado HLS [32]. This new driver has to be either built with the embedded Linux kernel [128] directly (for foreknown hardware) or with a device-tree overlay (for hardware known only at runtime). EDA tool vendors provide a hardware abstraction level (HAL) to make the development of the drivers easier via APIs to read and write to accelerators, to perform reconfiguration calls as well as basic C functionalities to enable debugging and fast development [127]. However, an accelerator developer or embedded Linux developer must take the responsibility to write and integrate the driver correctly into the rest of the Linux environment.

The layer above this fundamental driver and kernel layer forms 4) *a runtime system or libraries* for hardware acceleration. This provides a high-level API (such as OpenCL) to the user for serving reconfiguration and acceleration requests to the lower hardware layers [56, 119, 127]. However, currently, no FPGA vendor tools provide support for multi-tenancy.

Finally, 5) in the *application development stage* a developer (commonly a domain expert) performs the required task in software while using hardware acceleration where appropriate. Note that a developer at this stage may not possess the skills required to implement the underlying FPGA development stack.

Overall, each step of this stack requires sophisticated knowledge which makes designing systems challenging for individuals or small design teams. In case of large teams, these dependencies require frequent synchronisation and integration tests which slow down the development process. Given the complexity and the effort required for the task involved, often the final system is application-specific and cannot be reused or ported for different needs. To avoid this, we need a standard set of APIs such that a component can be swapped at each stage without recompilation or redevelopment of the components above or below it as long as the APIs are maintained. Thus, allowing each stage to be a modular artefact in itself.

Such an abstraction stack would also allow 1) the software stack to be reusable across all types of FPGAs and FPGA boards, 2) not require the accelerator developers to write drivers, and 3) update system components in the shell with no need to propagate the changes to other stages in the stack.

3.3.2 FOS Abstraction Stack for Modularity

We can achieve this by implementing 4 key layers of abstractions (as shown in Figure 3.4):

- **Decoupled Hardware Compilation Flow**: Ability to compile accelerators and shells in isolation from each other, i.e. the accelerators would compile against a fixed physical interface and a bounded resource region. This prevents any changes in IPs or the shell from propagating to accelerators (given that we do not change the PR region and its interface).

- **Logical Hardware Abstraction**: Exposing the accelerator and the shell in a high-level format with only a minimum set of parameters as required to build the drivers and perform the resource allocation. For the shell, this would include information such as how many regions it supports and at what addresses they can be accessed. Whereas, for the accelerators, it would include the i) internal hardware address register mapping (ADR map) for programming the accelerators and ii) meta-data associated with the accelerators for scheduling and management purposes (e.g., the size or maximum execution latency). Note that HLS compilation (through Vivado HLS) provides this information without the need of any manual step.

- **Accelerator Interface Libraries**: Using a standardised register mapping format to provide generic driver support for accelerators with streaming or master-slave interfaces (which are the most common interfaces provided by shells). Thus, it relieves the accelerator developer from the responsibility of writing drivers. In the case that the adoption of the standardised format is not feasible, a developer can use HAL APIs to build drivers.

- **Runtime and Multi-tenancy API**: Execution API which can abstract the interface at the logical level and which can span across multiple languages to provide high-level integration to existing software stacks, while supporting the necessary primitives or programming models for dynamic resource allocation.

Implementation details of these abstraction layers will be discussed in Section 4.2.

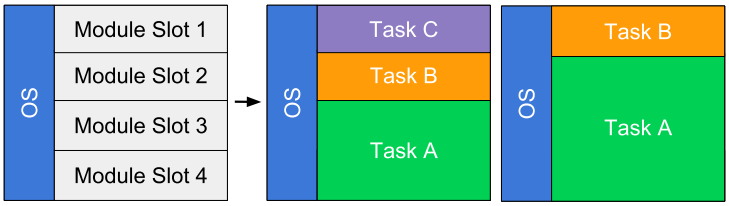


Figure 3.5: Physical FPGA layout featuring four partial reconfiguration regions (depicted as module slots). Hardware modules may take one or more adjacent slots and the communication is provided through a hardware operating system (OS) infrastructure.

3.4 FPGA Scheduling Concepts

An FPGA scheduler is responsible for adapting to the changing workloads and plays two main roles for virtualisation: 1) an illusion of unlimited FPGA slots (i.e. an application does not have to worry about currently available resources) and 2) optimise system utilisation and performance transparently from users. The following subsections describe the essential concepts required to achieve this.

3.4.1 Resource Elasticity

Let us first consider a software system where multiple tasks run in parallel on a single CPU. In this case, a scheduler would allocate the compute resources in the *time domain* to maximise the utilisation, i.e. assigning each task a time slot in which it can use the CPU. In contrast to this, an FPGA is a *spatial computing device*. Hence, a scheduler must allocate resources in the *spatial domain* to maximise the utilisation. This implies that reconfigurable resources must be allocated in a fractional amount to the tasks. We call this fractional block of reconfigurable resources a slot (or PR region). Figure 3.5 shows a visual representation of the slot. A scheduler can allocate one or more of these slots on the FPGA to tasks for FPGA sharing. In a dynamic system where tasks can arrive and leave at any time, the scheduler needs to be able to reallocate these slots to keep the utilisation high. To achieve this, the tasks on an FPGA must be '*resource elastic*' in terms of the number of slots it can operate on. We define resource elasticity as follows:

> ***Resource Elasticity:*** *the ability of a task to change its resource allocation transparently from the user. The change in resource allocation may be reflected in the performance (i.e. throughput or latency) of the accelerator used by the task.*

There are two main ways we can change the reconfigurable resources an accelerator uses: 1) **implementation alternatives** and 2) **replication**. With an implementation alternative, we can swap to another accelerator implementation which performs the same logical operation but with more or fewer resources for more or less performance in terms of throughput or latency of the operation. Note, an implementation alternative may provide *super-linear* performance benefit with respect to its resource requirements due to potentially a better algorithm, bigger local memory, loop tiling, unrolling or pipelining. Whereas with replication, we can ideally change the number of instances for a linear change in performance with respect to resources by controlling parallelism. This is analogous to changing the algorithm or the number of threads at runtime for performance optimisation of software applications.

Given a set of resource elastic accelerators, a resource elastic system can change the number of reconfigurable resources at runtime to adapt to changing workload as shown in the example scenario in Figure 3.6b. As shown in the figure, if only a single task is executing on the FPGA, the scheduler can allocate all the resources to it and, consequently, maximise its performance. Whenever a 'scheduling event' occurs, this allows to reduce or increase the system's allocation as appropriate by trading area for performance. By 'scheduling event' here we mean when 1) a new task arrives, 2) a task finishes, or 3) a context-switching point is reached. Details on context-switching points will be discussed further in Section 3.4.2. As shown in the example, changing resource allocation also causes some reconfiguration overhead. However, by dynamically growing and shrinking tasks, the scheduler can use free resources whenever available to accelerate the progress of executing tasks for achieving higher utilisation and faster execution (see Section 3.4.3 for trade-offs involved). Note, the resource reallocation can be performed transparently from the user by using context-switching mechanisms that involve partial reconfiguration of the FPGA.

Moreover, if the available slots cannot accommodate all tasks even when selecting the smallest implementation alternatives, a resource elastic scheduler can use time-division multiplexing (TDM) as a fallback approach. Analogous to a virtual memory subsystem using disk swapping for providing more than the physically available RAM at a performance penalty, TDM transparently allows the provision of more FPGA slots than physically available at some reconfiguration penalty.

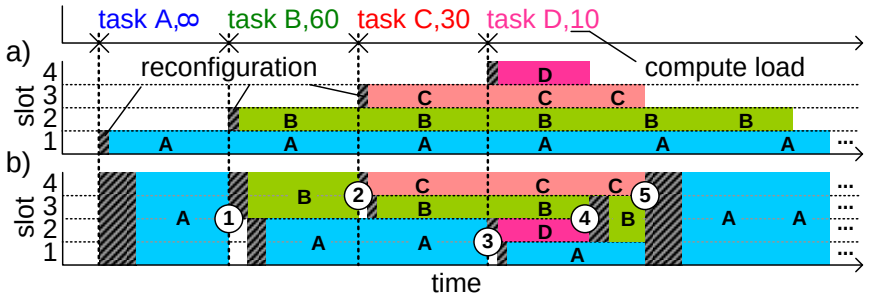


Figure 3.6: Resource allocation for tasks A, B, C and D in time when using a) Normal fixed module scheduling and b) Resource Elastic scheduling on a 4 slot FPGA. The circled events highlight cases where resources are needed to accommodate new arriving tasks (①, ②, ③) or cases where tasks complete (④, ⑤).

3.4.2 Hardware Context-Switch

In order to reallocate FPGA resources at runtime, we need to be able to perform context-switching of the hardware accelerators. Unlike CPU systems where we can use software instructions to store and restore registers to and from memory, for FPGAs, we have to pause and resume a hardware circuit with potentially arbitrary used storage elements like flip-flops, block RAM, latches and DSP registers. There has been much research dedicated to achieving this efficiently. The two primary research directions aiming at hardware context-switching are (as summarized in Table 3.1): system-specific and task-specific techniques. Each represents a distinct philosophy of implementing context-switching systems. System-specific techniques aim to perform context-switching by the system in a pre-defined standard method (typically using configuration read-back technique). In contrast, the task-specific techniques aim for letting the task perform the context-switch with its application-specific logic and in its format (e.g., by adding a scan chain into the accelerator modules). Consequently, the latency of context-switching is constant for the system-specific category and application dependent for the task-specific category. Note that partial reconfiguration needs to take place in all techniques to change the logic on the fabric, incurring overhead that ranges in milliseconds in addition to the application latency for the actual context-switch (saving-and-restoring the internal module state). Hence, keeping the context-switching latency minimal is very important for a dynamic system.

The commonly used system-specific techniques are 1) configuration read-back, and 2) fixed register map. For configuration read-back, we access the storage elements usable by the accelerators using the debug mechanisms provided on FPGA chips to read back the FPGA configuration, which includes the state of the accelerators. This

Table 3.1: Categorisation of context-switching techniques for hardware accelerators based on state storage format, restore latency and logic overhead.
* This is because system decides on max number of registers available for context save and restore.

Context-Switching Technique	Type	Latency	Logic overhead
Configuration Read-back	System-specific	High	High (logic constraints)
Fixed Register Map	System-specific*	Low	High (limited storage space)
Memory DMA	Task-specific	Medium	Medium (FSM for save-restore)
Scan Chain	Task-specific	Low	High (lack of BRAMs)
Data Parallelism	Task-specific	Low	Low

state can then be extracted and used to resume a module at a later point in time if required [39,58,76,98]. This imposes penalties equivalent to partial reconfiguration itself on most platforms (e.g., ReconOS [39] takes 25.7 ms for a region of 5616 LUTs on Virtex 6 and Knodel et al. [58,59] up to 1 sec for a region of 30800 LUTs on Virtex 7) and restricts the type and placement of storage elements. For example, Xilinx FPGAs feature block RAM and multipliers (DSP) primitives which can provide pipeline registers. These registers are, however, not accessible using built-in debug mechanisms of the FPGA. The alternative fixed register map technique provides a fixed set of memory-mapped registers inside an accelerator where it can store its relevant state like software systems [105]. The host CPU can then save and restore these memory-mapped registers for context-switching. This approach can work well for soft-cores and overlay accelerators. However, for other dataflow type accelerators, it can impose considerable limitations and overheads on the accelerator design.

There are three main task-specific context-switching mechanisms: 1) memory DMA, 2) scan chain, and 3) data parallelism. The memory DMA technique relies on adding additional logic in the accelerator to provide access for storing and restoring its state to memory when requested for context-switching [105,120]. Thus the latency of context-switching depends directly on the memory transfer required by the accelerator. It also requires additional control logic in the accelerator to be able to perform the transfer, which may become a critical path or make the accelerator resource bound. The second approach is using scan chains where we instrument an accelerator to include all necessary memory elements in an alternative shift register mode for state access [11,61]. This imposes the penalty in milliseconds for context-switching.

The third approach of task-specific context-switching is to rely on data parallelism, i.e. perform context-switching only at the end of a parallel data chunk. It avoids the

need for storing and restoring the state entirely. However, it cannot provide preemptive context-switching capabilities on its own, as the context-switching points depend on the execution latency for processing a parallel data chunk. Although, we can combine it with the other context-switching techniques to provide preemptive context-switch if required. Note that we can easily find data-parallelism for most common FPGA applications, i.e. streaming and batching type FPGA applications. Examples of such applications includes image processing, neural networks, and cryptography.

In summary, the system-specific techniques are cheap (in terms of resources) and can be performed eventually automatically but slow and with some restrictions. In contrast, task-specific techniques are fast at the cost of extra logic. The suitability of the techniques depends on the application and non-functional properties.

Minimising the relevant state space can reduce the context-switching latency of the above methods. In particular, employing context-switching at only selected execution points (cooperative context-switching), where the internal state space is minimal for an application, can further improve performance [43]. For example, in a Convolutional Neural Network (CNN), context-switching may only be performed at the frame border where the internal line buffer does not carry a state needed in the future. Note, these minimal state points should be accompanied with appropriate handling of pending I/O transactions without any loss of data. Hence, there is a trade-off between scheduling flexibility and overhead for context-switching when selecting between preemptive context-switching and cooperative context-switching.

Overall, for implementing resource elasticity, we cannot directly apply techniques like scan chains and configuration read-back, as they tie the state encoding to the accelerator design. This does not easily allow a system to change to implementation alternatives without state transformation in the case that resource elasticity is using implementation alternatives. Whereas memory DMA, data parallelism, and fixed register map allow directly switching to implementation alternatives, as the accelerator is in control of how storage elements are used *inside* the accelerator. However, without data parallelism, it is not possible to change the number of accelerator instances dynamically. Hence, we use data parallelism as means of context-switching with cooperative scheduling to achieve resource elasticity in both forms (implementation alternative and replication) at minimal context-switching overhead. This can alternatively be combined with memory DMA for applications without data parallelism to improve the scheduling flexibility if need be for a more holistic solution.

3.4.3 Trade-offs

A resource elastic scheduler (RES) must perform three main trade-offs which are not commonly considered by standard time-domain schedulers:

1. **Multiple instances vs Different sized modules**: At runtime, if we can allocate more resources to the task for better performance, we need to decide whether to do so by replicating an accelerator or by switching to an implementation alternative (Trade-off 1). We need to consider that the penalty of pausing the currently running module and performing partial reconfiguration for changing to a different-sized module may not be the best option given the work remaining and the possible speed-up achievable with a different-sized module. Figure 3.7 shows an example scenario with allocation alternatives which a resource elastic scheduler must choose from, depending on the aim.

2. **Run to completion vs Changing module layout**: We must decide if it is worth to change the resource allocation for a task given the amount of work left and the partial reconfiguration overhead it may cause (Trade-off 2). This holds regardless if we use reconfiguration for shrinking and expansion of accelerators or for defragmenting the FPGA slots. In this case, we must decide on whether changing the resource allocation to achieve better system-level performance at the cost of performance-sacrifice for certain modules is beneficial or not.

3. **Collocated change vs Distributed change**: The decision of selecting a multi-instance option over a different module size may also depend on the location of available free slots and the implementation alternatives available for the tasks (Trade-off 3). For example, consider the scenario as shown in Figure 3.8 where the only available resource slot is at one end of the FPGA. In this scenario, the possible options for maximising resource utilisation for Task A are to either replicate A or to defragment the FPGA such that we collocate available slots to use a bigger implementation alternative for A.

Note, the scenarios used in the above description of trade-offs are relatively simple as the focus is just on Task A. The complexity of the possible situation increases as we consider multiple distinct tasks where it may be necessary to sacrifice performance for a particular task to accelerate another for achieving higher overall system performance. Further, the aim of the scheduler may not be performance but fairness or energy or quality of service, in which case, scheduling policies need to decide accordingly.

Figure 3.7: Logical slot configuration example where a) shows using multiple instances of a single slot module, while b) shows using different sized module, which may have super-linear speed compared to a single slot module.

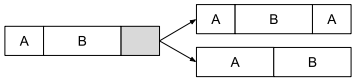


Figure 3.8: Example scenario where there are two possible alternatives. 1) Replicate A or 2) perform defragmentation to use different sized module for A.

At first glance, a slotted architecture (as shown in Figure 3.5), gives the impression that resource elastic scheduling is similar to the multi-core scheduling known from software systems. However, FPGA scheduling has to consider several very different aspects such that i) an accelerator may occupy multiple adjacent slots simultaneously and cause fragmentation issues, ii) context-switching is significantly more expensive, hence, should be used more infrequently and iii) each implementation alternative has its own performance behaviour and should be applied based on workload.

Overall, this makes the resource elastic scheduling more complex than the standard strip packing problem and more similar to the resource-constrained project scheduling problem which is known from the field of Business and Operation. Appendix A contains the theoretical formulation of this resource elastic scheduling problem for FPGAs and its mapping for a heterogeneous system.

Section 4.2.5 and 4.3 will further discuss the implementation details of resource elastic schedulers.

3.5 Heterogeneous CPU+FPGA Systems

Modern FPGA systems from industry are moving towards a unified platform architecture for both datacenter and embedded FPGA devices. An example of an industry platform is the Xilinx RunTime system (also called XRT) [126] as shown in Figure 3.9, where we can see an embedded system featuring a hardened ARM CPU on-chip while for datacenter systems, a soft-CPU is used to program and manage FPGA accelerators on behalf of the host machine. The soft-core on datacenter FPGAs primarily exists to avoids saturating PCIe bandwidth with small control transactions but can also be used to provide other means of abstracting I/O and management for accelerators [16,75,99]. Hence, regardless of being an embedded system or datacenter setup, we can assume that there exists a CPU (hard or soft) on-chip along with the FPGA fabric, capable of

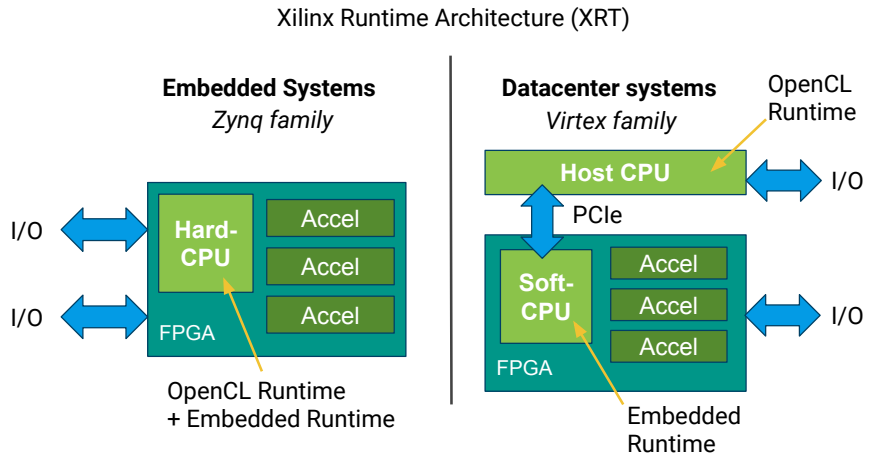


Figure 3.9: Xilinx RunTime system (XRT) architecture.

accessing memory and of performing control-oriented tasks efficiently.

However, this CPU is often not used for acceleration and, in cases where it is, the decision of what work it should perform is left to the user. In particular, even advance heterogeneous systems with OpenCL runtimes for *both* CPU and FPGA, the execution is tied to an unchangeable set of resources and is only available for a single application at a time [37]. This restricts the potential performance achievable when using heterogeneous resources which can accelerate both control-flow and data-flow type of applications efficiently. To maximise the performance and utilisation of such heterogeneous systems, a new paradigm of resource management is required which can support multiple applications while optimising the four main scheduling problems:

1. **Device type selection**: Which device (or a combination of devices) an application should be executed on (i.e. deciding between CPU or FPGA)?

2. **Workload partitioning**: If scheduling on multiple devices, how much workload should be executed on each device? E.g., scheduling 25% of the threads on the CPU and 75% of the threads on the FPGA.

3. **Number of compute units**: How many instances of compute units should be allocated for a task? E.g., the number of CPU cores or FPGA accelerators.

4. **Accelerator type selection**: If a device has multiple *implementation alternatives* (e.g., FPGA accelerators with different micro-architectures or big.LITTLE CPU cores), which implementation should be selected?

Note, here and onwards in this thesis, the keyword '*device*' refers to either the CPU or the FPGA on-chip and the keyword '*compute unit*' refers to an execution unit on a device, i.e. CPU cores and accelerators running on an FPGA.

Current runtime systems tend to solve only a subset of these problems for a single application using profiling information (or manual programmer instrumentation) for deciding the appropriate device type, workload partitioning, and number of compute units [37, 46]. These systems further refrain from context-switching on FPGA accelerators and treat an FPGA fabric as an ASIC accelerator, ignoring the reconfigurability of FPGAs entirely. This is because context-switching is a very expensive operation [39, 61] (see Section 3.4.2 for further details).

Hence, to tackle this, we first propose using cooperative scheduling for FPGAs to lower the cost of context-switch (as mentioned in Section 3.4.2). Secondly, this is combined with an abstraction layer to treat both CPUs and FPGA equally with respect to the execution model. An example of such a model is OpenCL (see Section 3.5.2 for details). This abstraction layer allows us to execute an application on either CPU or FPGA without changes in the source code. With the ability to context-switch on *both* CPU and FPGA and move computation between devices (will be discussed in the next subsection), we can consider arbitrarily reallocating heterogeneous resources at runtime. In particular, this allows us to change the device and accelerator type selection as well as the number of compute units at runtime. When applied with dynamic workload partitioning and heuristics for over-committing resources, we can tackle the four scheduling problems together and enable collaborative execution on all CPU and FPGA resources transparently from the user while maximising system utilisation. We call this ability to change the resource allocation dynamically for a task: *'heterogeneous resource elasticity'*.

3.5.1 Runtime Device Type Reallocation

From a conceptual point of view, we can also perform resource elastic scheduling for CPUs by treating it as another form of compute device with its own performance and resource requirements. However, to move a task dynamically from/to CPU (software version) to/from FPGA (hardware version), we need to able to perform device type reallocation, i.e. pause a hardware accelerator via a context-switch and have it resumed in software via context-switch or vice versa. There are three main ways we can achieve this (as shown in Figure 3.10):

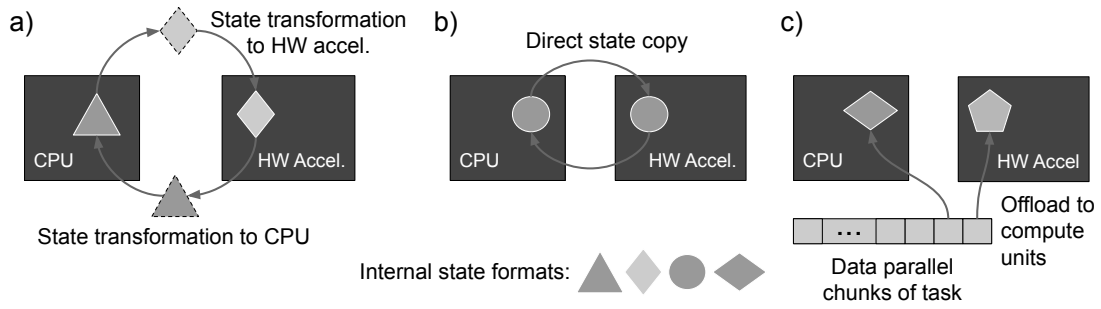


Figure 3.10: Three ways to perform application migration across device types, where a) depicts method transforming the state at runtime across devices, b) represents technique of common encoding format, and c) shows the data parallelism approach.

1. By transforming the hardware state into software state encoding at runtime.
2. Having a standard state storage encoding for both hardware accelerators and software implementation.
3. Exploiting data parallelism available for an application such that no internal state needs to be preserved when switching between CPU and FPGA.

Transforming the state encoding at runtime can potentially be arbitrarily complex and can impose additional latency even if a specialised hardware unit performs it [62]. Further, the approach can only be applied to a subset of applications, as the runtime system would then need to be aware of differences in state encodings and their transformation across all device implementations. This makes the approach non-scalable for general purpose use. Having a common state encoding as a system standard can avoid this issue and allows seamless arbitration from software to hardware and vice versa with an appropriate context-switching mechanism. However, enforcing the hardware to store its internal state in a given format may restrict accelerator design and optimisation which would have been otherwise possible. Hence, trading application performance with device reallocation overhead. In contrast, exploiting the data parallelism available in most FPGA applications can remove the device reallocation overhead as well as the restrictions on accelerator design. Although this will limit the type of applications the system can support. However, given that i) most applications follow a batching and streaming model for FPGA acceleration, ii) data parallelism's compatibility with the context-switching model for FPGA, and iii) have the ability to perform collaborative execution based on data independence, we adopt this method as our hardware-software mechanism for reallocation of resources across device types (CPU and FPGA).

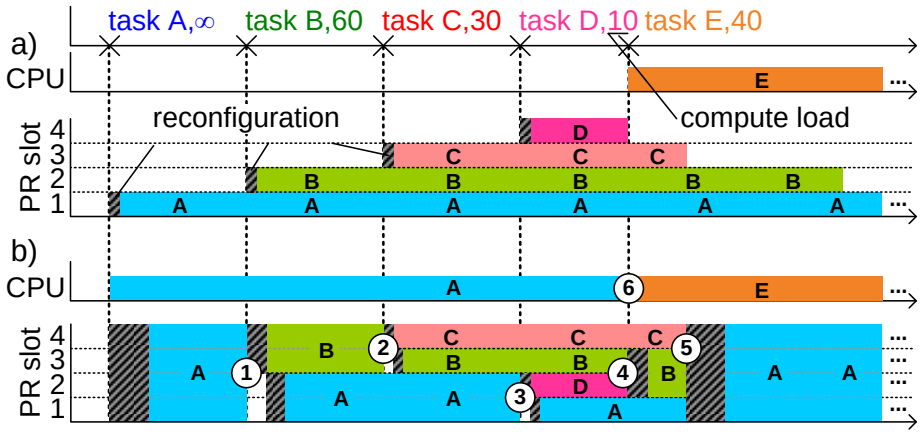


Figure 3.11: Resource allocation for tasks {A-E} in time when using a) round-robin scheduling and b) heterogeneous resource-elastic scheduling on a CPU+FPGA architecture. The circled events highlight cases where resources accommodate newly arriving tasks (①, ②, ③, ⑥) or cases where tasks complete (④, ⑤).

Figure 3.11 shows an example execution trace of a system which can migrate tasks across devices in addition to resource elastic scheduling on the FPGA. Whereby when a single task is executing on the system, it will use both CPU and FPGA resources for execution. Upon arrival or finish of other tasks, the runtime system adjusts the resource allocation without the user realising by using a common context-switching mechanism (e.g., data parallelism in our case).

3.5.2 OpenCL Execution Model

OpenCL is one of the most common industry standards for the execution of data and task-parallel applications on heterogeneous devices such as CPUs, graphical processing units (GPUs), FPGAs, and domain-specific processors (DSPs). OpenCL specifies an execution model and a set of APIs to control and enable programming of these devices. In particular, OpenCL allows compiling the same source code for different devices to improve the portability and programmer productivity [77].

An OpenCL application has two components: a *host program* and *kernels*. A host program manages memory objects and issues execution commands while executing on a host machine, whereas kernels are compute-heavy functions which execute on an accelerator's compute unit. When a host program issues a kernel execution command, an abstract index space known as *NDRange* is generated. The kernel function executes once for every point in this NDRange index and is known as a *work-item* (can

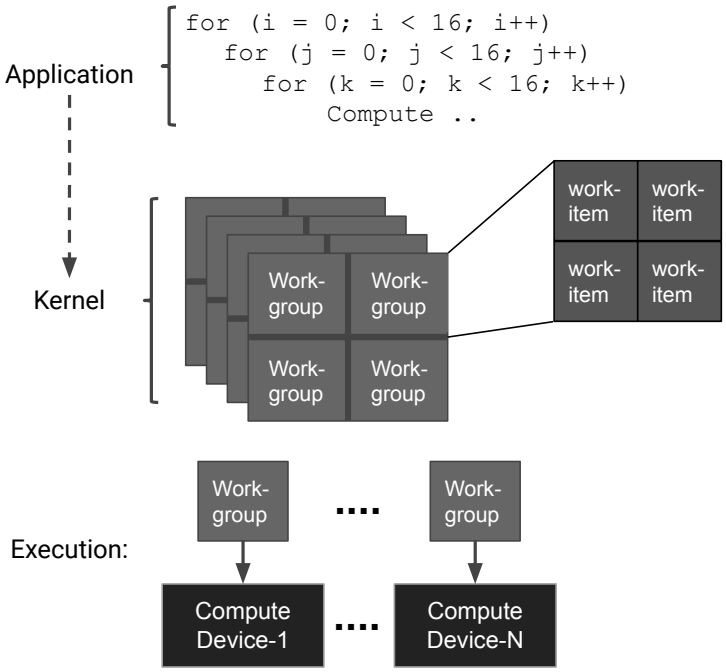


Figure 3.12: Structure and execution of OpenCL kernels on computer devices.

be considered analogous to a thread). These work-items execute in a group on a computing unit; this group of work-items is called a *work-group*. Figure 3.12 shows the visual representation of the kernel. A work-group provides a coarse decomposition of NDRange and allows work-items to share and synchronise data using barriers on local memory. However, there is no execution order defined between work-groups by the OpenCL standard, which allows them to execute concurrently on different compute units for high performance [77]. At the end of the work-group execution, the result is written back to the global memory without any synchronisation.

Since there is no execution order defined and all the state information is stored in memory at the end of work-groups, we can perform a data parallelism context-switch for OpenCL applications cooperatively as mentioned in Section 3.4.2. With this, we can *at the end of each work-group* change to another implementation alternative, pause the execution and resume later, or run the kernel concurrently on multiple hardware accelerators as well as CPUs.

Implementation details of an OpenCL runtime system with heterogeneous resource elastic scheduling will be discussed in Section 4.3.

3.6 Transparent FPGA-to-FPGA Accelerator Migration

With this large-scale deployment of FPGAs in distributed systems, it has become important to provide solutions for fault tolerance and high availability of these acceleration services. In particular, migration is essential in modern datacenters for three primary use cases:

- **Maintenance**: Allowing to continue delivering a service while a datacenter node is upgraded. With the help of migration, the application can be temporarily moved to a remote node while the upgrade is performed transparently.
- **Resource management**: Migration enables dynamic load-balancing to redistribute work as the workload changes for multi-node systems.
- **Fault tolerance**: In case of a fault, an application needs to be migrated to another node where it can resume execution from the last known consistent state (using check-pointing).

Standard software systems employ live virtual machine migration with distributed check-pointing mechanisms to meet these requirements [106]. Conceptually, for FPGAs this would translate into migrating a hardware task from one FPGA to another with a potentially different number of resources available, using partial reconfiguration and state transfer over the network (as shown in Figure 3.13). However, for FPGAs, these techniques known from the software systems cannot be applied directly as the application running on FPGAs represent hardware circuits rather than a sequence of instructions. A fully transparent preemptive context switch for FPGAs is expensive in terms of both latency and design overhead. Hence, we need a coarse-grained approach of cooperative context-switch (as explained in Section 3.4.2).

Given the ability to perform a context switch, we can perform the migration by pausing the hardware accelerator on a source node and transferring the necessary state and data information to a target node and then resuming the execution again on the target node (as shown in Figure 3.14b). We call this process blocking migration, whereby the computation stops while the state information is being transferred between the nodes. Similar techniques are employed for GPUs [121] using OpenCL, where an OpenCL kernel is broken down into sub-kernels (a group of work-groups) and executed one after the other. This allows to pause and resume the application execution at the end of sub-kernel execution for migration purposes. However, the blocking nature of the process represents the major limitation, as the overhead becomes directly

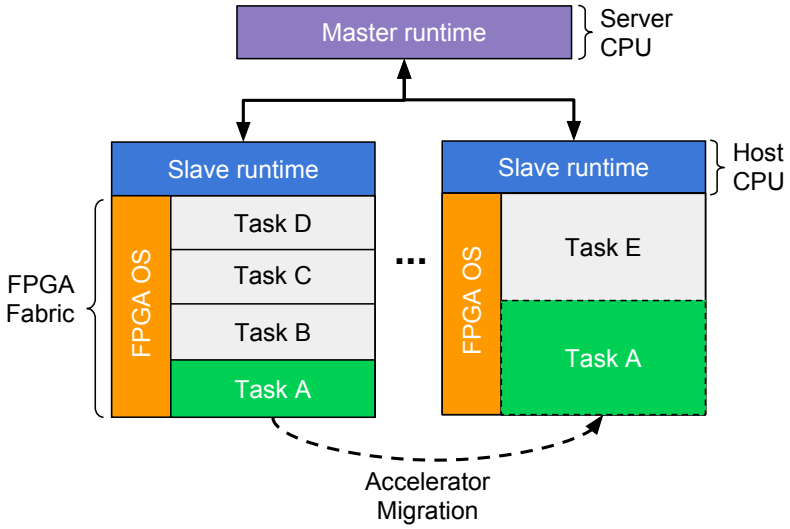


Figure 3.13: Migration of FPGA accelerators.

proportional to the amount of data transfer required for the application.

Hence, we propose an alternative non-blocking approach that relies on the data-parallel nature of the OpenCL execution model (see Section 3.5.2) to overcome this limitation. Given the assumption that the application does not overwrite the input data during the execution[1], it is possible to transfer input data during the execution on the source node along with necessary control information and bitstream of the accelerator to the target node. Further, as the execution order of different work-groups is undefined, the target node can start execution of the next work-group before the source node completes its work-group execution. In detail, the algorithm works as follows (while Figure 3.14c shows the resulting execution trace):

1. Transfer the accelerator bitstream and the input data while continuing execution on the source FPGA.
2. After initialisation of the accelerator on the target FPGA, stop issuing new work-group to the source FPGA accelerator.
3. Start issuing new work-groups to the target FPGA accelerator.
4. After last work-group execution on the source FPGA, transfer the output data to the target FPGA.
5. Continue execution of remaining work-groups on the target FPGA and perform data merging on the output data in the background.

[1]For fault-tolerant systems, this is a norm to enable recovery and re-execution.

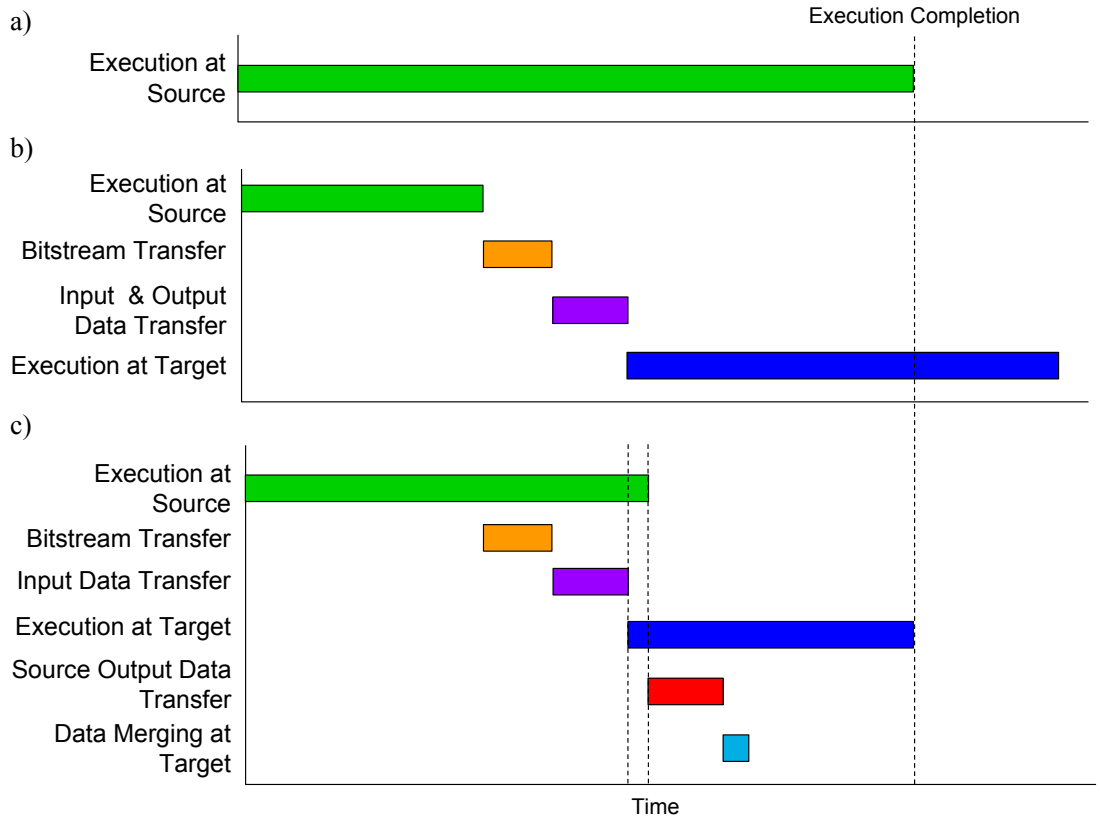


Figure 3.14: Execution trace of accelerator where a) is no migration case, b) is the blocking migration case and c) is the non-blocking migration case.

Implementation details of the proposed migration methods will be discussed in Section 4.4.

3.7 System Requirements and Chapter Summary

To build the solutions proposed in this chapter, we need to meet the following three requirements:

1. An FPGA shell (static system) infrastructure which can provide four main features to support resource elastic hardware accelerators with modularity:
   (a) **Decoupled compilation flow**: to compile shell or accelerators in isolation from each other, such that changes in the shell do not need to propagate to the accelerators.
   (b) **Multiple partial regions**: to change resource allocation of tasks without interrupting other concurrently running tasks.
   (c) **Variable sized partial regions**: to support implementation alternatives with different reconfigurable resource requirements.
   (d) **Relocatable modules**: to keep the number of bitstreams minimal. This is a desirable feature because the standard partial reconfiguration flow requires to generate a bitstream per region for each implementation alternative, which can potentially be expensive for multi-tenant systems. Note bitstream size range in tens to hundreds of mega-bytes for Xilinx Ultrascale+ FPGAs.
2. A unified task description model which can compile to both software and hardware, such that, the user need not be aware of which implementation is used during execution except for the difference in performance, to aid the programmability of the heterogeneous system-on-chip.
3. Applications which are implemented with data-parallelism to allow dynamic scaling the number of compute units (i.e. CPU cores or FPGA accelerators). This also allows performing the context-switch with least overhead on FPGAs for dynamic resource allocation.

3.7.1 Chapter Summary

Overall, in this chapter, we proposed 1) using a modular development flow to allow system components of an operating system to be changeable and portable. We then advocated 2) combining the modular FPGA OS with resource elasticity to allow changing the resource allocation in the spatial domain to maximise utilisation. 3) To support this, we recommended using cooperative context-switching to avoid the high latency of preemptive context-switching on FPGAs. In particular, we use the data-parallelism already present in most FPGA applications to perform context-switching by only adding partial reconfiguration latency. Further, 4) to maximise the utilisation of CPUs along with FPGAs, we propose dynamically changing the device and accelerator type, the number of compute units and workload partitioning while serving multiple users. This is achieved by using a unified execution model between different device types and data-parallelism for context-switching between these devices. Moreover, 5) we presented methods to perform live migration for OpenCL FPGA accelerators in blocking and non-blocking manner across different FPGA nodes with low overhead to support maintainability, load-balancing, and fault tolerance in multi-node FPGA systems.

The implementation details of the methods proposed in this chapter can be found in Chapter 4 while their evaluation based on varying workloads, accelerator behaviours, platforms and use cases can be found in Chapter 5.

Chapter 4

Building Modular Resource Elastic Systems

4.1 Chapter Overview

In this chapter, we describe the implementation details of the concepts and methods described in Chapter 3 to improve maintainability, adaptability, and accessibility of FPGA systems. This is carried out in three parts (see Figure 4.1). First, this chapter describes how to build a modular FPGA platform with easy-to-use interfaces. This is followed by how to allocate resources dynamically to maximise CPU+FPGA system utilisation. Finally, the chapter ends with how to implement live migration using a modular system and dynamic resource allocation to enable better support for FPGA clusters. An overview of each part is as follows:

1. **Modular FPGA Operating System (FOS)**: This comprises the development of an FPGA shell and the software layers required to achieve a modular FPGA development stack with existing tools. In particular, it provides using the FPGA in multiple modes from the static acceleration to the multi-tenant environment, with easy to use APIs for both software and hardware developers. The libraries and scheduler of FOS, which enable this wide range of use cases, aim at serving generic workload with no prior knowledge of accelerator behaviour. Consequently, the default FOS scheduler does not necessarily solve the resource elastic scheduling problem optimally. As an online management method, it uses greedy heuristics to provide fairness between users, and it speeds up execution when there are free resources available. To build FOS, the FPGA shells developed by

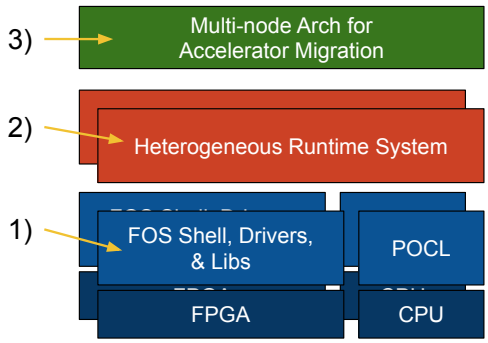


Figure 4.1: Implemented system with 1) a modular FPGA operating system (FOS) which will be described in Section 4.2; 2) heterogeneous runtime system which will be described in Section 4.3; and 3) a multi-node architecture for accelerator migration which will be presented in Section 4.4.

K.D. Pham [85, 86] has been used in order to meet system requirement 1 (i.e. a shell supporting modularity and flexibility, see Section 3.7), while the contribution of this thesis is the development of the rest of the FOS components and abstraction layers. FOS is freely available as an open-source project on GitHub[1].

2. **Heterogeneous Runtime System**: It is built on top of FOS and replaces FOS's generic scheduler to provide better dynamic resource allocation for CPU+FPGA systems using OpenCL as a unified task description model. The adoption of OpenCL allows meeting the system requirement 2 and 3 (i.e. common execution model and application parallelism, see Section 3.7) while catering for one of the most common workload types for both FPGA and heterogeneous system in the industry. Since resource elastic scheduling is a mixed-integer non-linear programming problem with higher complexity than strip packing problem (see Appendix A for the theoretical problem statement), the runtime system uses a Branch-and-Bound algorithm to solve resource elasticity trade-offs and all four optimisation problems of heterogeneous resource management (mentioned in Section 3.5). An alternative brute-force algorithm implementation targeting performance and fairness on FPGA-only systems can be found in Appendix B with an evaluation for long and short running task workload.

3. **Multi-node Architecture for Accelerator Migration**: It is built on top of the resource elastic runtime system with FOS as its base, in order to accept OpenCL acceleration requests on FPGA clusters. The primary goal for this system is to

[1] https://github.com/khoapham/fos

demonstrate the use and evaluation of live migration techniques, as described in Section 3.6 to enable maintenance, load-balancing, and fault-tolerance in multi-node FPGA systems.

4.2 FOS: Modular FPGA Operating System

In this section, we describe the implementation of the FOS with the concepts introduced in Section 3.2, to improve the maintainability of FPGA systems and the portability of its system components.

4.2.1 FOS Overview

FOS is a modular and lightweight FPGA Operating System which provides three primary modes of operation (as shown in Figure 4.2):

1. Execution on static accelerators in a single-tenant mode.
2. Using multiple partially reconfigurable accelerators in a single-tenant mode.
3. Dynamically offloading acceleration request in a multi-tenant mode.

To provide the first two-modes of operation with ease of access, FOS provides two libraries: Cynq and Ponq. They provide high-level APIs to interact with hardware used from C++ and Python applications, respectively. These *APIs are platform-independent* and can support the traditional Xilinx PR flow as well as the decoupled compilation flow introduced with FOS.

The multi-tenant mode is provided through a daemon, which orchestrates the hardware acceleration requests from different users and schedules them in time *and* in the spatial domain. Moreover, the daemon can execute hardware acceleration requests on heterogeneous accelerators (i.e. accelerators written in different languages) concurrently while providing the same user interface.

To support the underlying hardware interaction in each mode, the libraries include generic drivers to program accelerators and the Xilinx FPGA manager [128] for partial reconfiguration. Moreover, FOS uses the PetaLinux Kernel [128] and Ubuntu rootfs, to adopt the standard functionalities provided by the Ubuntu Linux distribution such as access to file systems, network access (via host-CPU), standard libraries, debugging tools, and other forms of I/O.

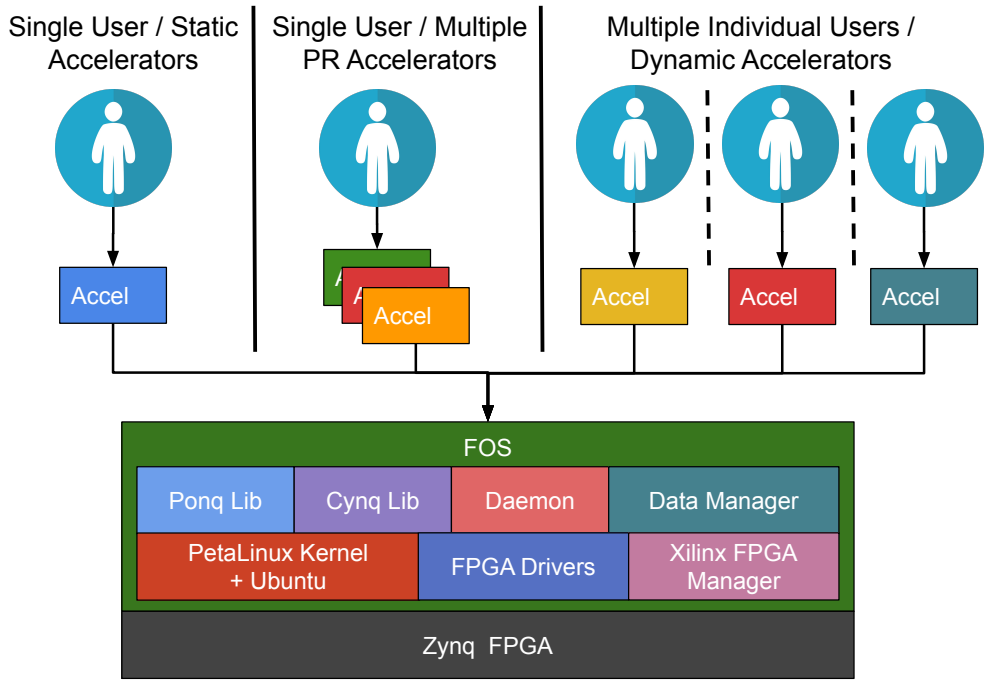


Figure 4.2: System components and usage modes of FOS – FPGA Operating System.

The hardware infrastructure used in this thesis is based on the open-source ZUCL 2.0 shell [85]. It currently supports three boards of varying FPGA capacity: ZCU102 (a Xilinx MPSoC development kit), UltraZed, and Ultra-96 (suitable for IoT and edge deployment) boards. The shell provides the ability to reallocate hardware accelerators across different partial regions as well as to combine multiple adjacent partial regions to host bigger accelerators.

With these additions to the software and hardware ecosystem for FPGAs, FOS achieves a similar level of support for rapid development, deployment, and ease of use, as known from standard operating systems for CPUs.

4.2.2 Decoupled Compilation Flow for Shell and Modules

The primary requirement for abstracting accelerators from the shell is to decouple their compilation. However, this alone does not enable the flexible resource allocation required for the maximum utilisation of the resources. This is because for flexibility, multiple partial regions should be combinable for hosting different sized modules and modules should be relocatable and duplicatable (at bitstream level). However, none of these requirements are met when using standard vendor EDA tool flow. To solve this, there are strict requirements in order to achieve isolation in development process with

system support for relocation and partial region flexibility:

1. **PR regions should be homogeneous** in terms of their resource foot-print (i.e. the relative layout of FPGA primitives) to allow accelerator relocation, and regions should be adjacent to each other to allow hosting bigger accelerators without interfering with other system components.
2. **Communication interfaces between modules and the static system must be identical** in terms of both logical protocol and their physical implementation such that relative positions of connection wires are the same in all PR regions. This ensures that modules can receive operation commands from the host CPU and it provides interfaces to transfer data back and forth to the main memory, irrespective of a module placement position.
3. **Clock signals must follow the same regular pattern across every PR region**. These constrained clock routing paths which will be used to provide clock signals to the modules regardless of the final target position on the FPGA.
4. **Routing from the static part must be prohibited from passing through the reconfiguration part and vice versa** (except the interface routing) to ensure that module relocation does not interfere with other parts of the system[2].

The here introduced decoupled compilation flow relies on the standard FPGA development flow (Vivado tool-chain for Xilinx FPGAs [124]) in order to implement 1) the basic infrastructure, which acts as the OS shell, and 2) the hardware applications. However, the process is guided with additional design steps, i.e. TCL constraints and academic tools to adapt and customise the default flow to build the final system, as shown in Figure 4.3. This basic compilation flow is provided by K. D. Pham [83] as a part of this PhD project while the author has conducted the testing and deployment (i.e. its automation and drivers for accelerator operation). Consequently, the details of the flow are only included here for completeness.

The main steps of the compilation process are as follows:

1. **Planning**: In this step, a shell developer needs to make a series of system-level design decisions, including:
   (a) **Resource partitioning**: Based on the FPGA fabric layout and the number of resources available, a shell developer needs to split the FPGA fabric into

[2]This is not a strict requirement and the tool GoAhead [6] used to implement the shell can be used in a mode that allows static routing to cross reconfigurable regions.

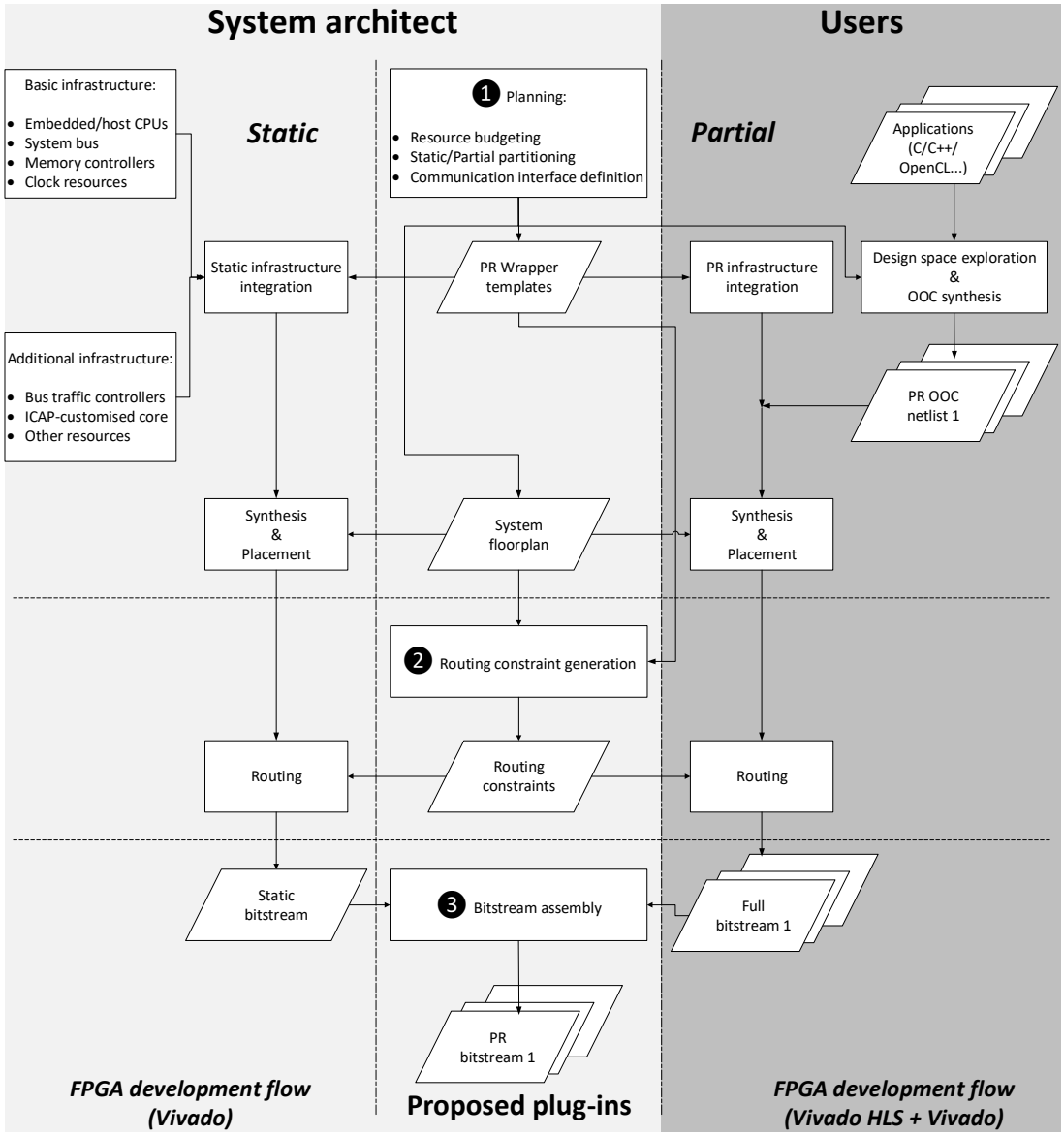


Figure 4.3: The decoupled compilation flow with support for relocation and variable size modules. The left part is performed once for a specific version of a shell by a shell developer while the right part is performed once for each module by FOS users [83]. Modules do not have to be aware of the shell and are compiled in out of context (OOC) mode.

two parts: i) the static region for the FPGA shell and ii) one or more reconfigurable regions for hosting hardware accelerators. This also defines the size of each PR region on which the High-Level Synthesis (HLS) tools [32] can perform the Design Space Exploration (DSE) for throughput optimisation [78]. The trade-off to consider at this stage requires the developer to allocate the maximum amount of resources to the reconfigurable part while leaving sufficient resources to the static part for shell functionalities and future upgrades. This trade-off is system-dependent and requires a thorough understanding of the target FPGA device and system requirements. However, this step is only performed once for a particular system. The result of this process is 1) the static system floorplan and 2) bounding boxes of the reconfigurable regions.

(b) **Communication interface definition**: Selection of the protocol and data-widths for communication between the static system and the partial regions based on the system requirements.

2. **Routing Constraint Generation**: The system floorplan and the PR interface template from the previous step are used to generate routing constraints. We can describe the implementation rules with the help of academic PR frameworks (GoAhead [6] and a TCL library [111]) in the form of TCL files for routing constraints automatically. These TCL constraints will then guide the routing stage of Vivado.

3. **Configuration Bitstream Generation**: The proposed flow results in full *static bitstreams* for both shell and module designs. To compose *partial bitstreams* for accelerator modules from these bitstreams, we use the bitstream manipulation tool BitMan [84].

Shell Development

The static design starts with integrating basic infrastructure IP and additional infrastructure to a top-level unified design. In our designs the basic infrastructure includes 64-bit ARM Cortex-A53 CPU cores, AXI4 interconnects, memory controllers, Xilinx PR Decouplers for disabling/enabling static and module communication, clock management tiles for tuning module frequencies, and PR Module Interfaces. It is possible to add other resources, such as memory management or network communication IPs, if required.

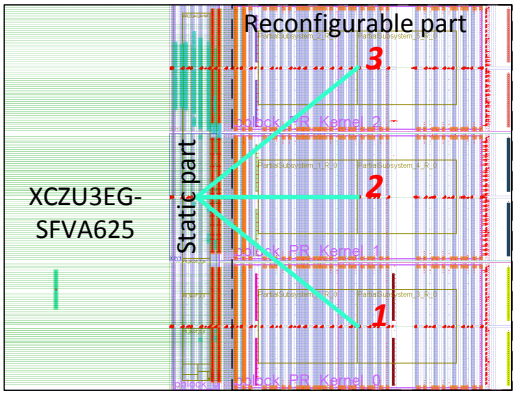


Figure 4.4: The physical shell implementation on the UltraZed and Ultra96 boards. This version has *three slots* and can host up to three FPGA applications simultaneously.

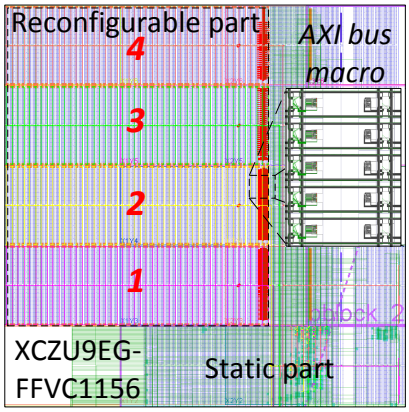


Figure 4.5: The physical shell implementation on the ZCU102 board. This version has *four slots* and can host up to four FPGA applications simultaneously.

The PR Module Interface provides an AXI4-Lite slave for control register access via the CPU and an AXI4 master for memory access. This fixed interface between the hardware module and the static system is implemented using the available routing resources in the Zynq UltraScale+ FPGA devices. In particular, we keep this interface identical for all PR regions to serve relocatable hardware modules by pre-placing and pre-routing these communication signals in a constrained, predefined manner. This reassembles the bus macro approach, which was very popular for Xilinx Virtex-II devices [68]. However, in the here used flow, this can be implemented without logic cost (in terms of LUTs used for the bus macro).

To keep the clocking resources of PR regions identical for relocatable hardware modules, we block all routing wires except for a defined subset of the `BUFCE_LEAF` primitives inside the PR regions. This forces the router to use only a defined subset of these clock driver primitives that each drive a specific vertical clock spline which ultimately connects to the flops, BRAMs, and DSPs in the region. However, we only route the clock for the PR regions this way. This means when routing the static system, we 1) route the PR module clocks with `prohibit` constraints on the `BUFCE_LEAF` primitives, then 2) remove these constraints, and 3) incrementally route the rest of the system. This allows routing additional clock nets as needed by the static system (e.g., for providing clocks to memory controllers or gigabit transceivers).

Finally, to prevent any static signal from violating PR regions, we insert a blocker

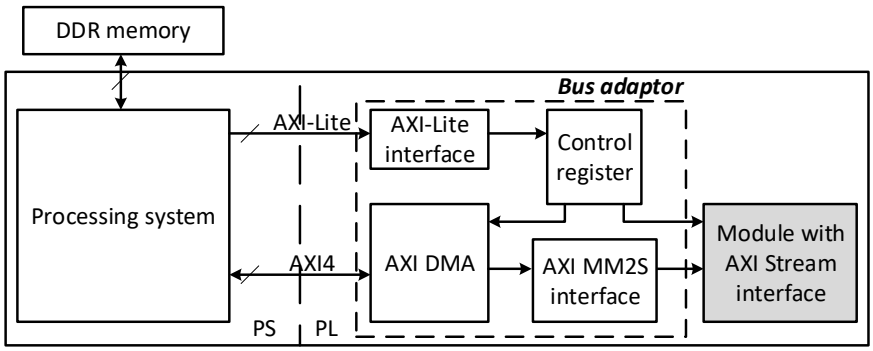


Figure 4.6: An example for bus virtualisation: the module has a 32-bit AXI-Lite interface and a 32-bit AXI Stream interface without DMA engine. In this case, the *bus adaptor* with AXI DMA and AXI MM2S IPs are chosen to carry out the communication with the rest of the system.

macro. This blocker is non-functional but uses all local wire resources inside the reconfigurable regions before routing the static system. We generate this blocker macro using GoAhead [6] or the TedTCL library [111] according to the system floorplan.

The above steps result in the final static physical implementation as shown in Figure 4.4 for the UltraZed/Ultra96 board and in Figure 4.5 for the ZCU102 board. We can see that the PR region interfaces have the same relative physical positions and the distribution of clock splines across all PR regions is identical. Both systems support combining multiple adjacent regions for hosting larger monolithic modules. In this case, we only use one PR module interface.

Bus Virtualisation

Operating a hardware accelerator needs communication with the host CPU to issue commands and access to DDR memory for the actual data processing. In the here implemented system, a module can use a wide range of bus widths (such as 32/64/128-bit width) and various bus protocols (such as AXI4 Master/Stream). In particular, HLS modules, often by default, include DMA engines for fetching data from memory and for writing back results. However, this is not always the case with hand-crafted RTL or customised netlist accelerators which may operate on data streams rather than data stored in DDR memory.

We tackle this issue by providing another level of abstraction for bus interfaces between the FPGA applications and shells. For this, we selected the interface to provide the 32-bit AXI-Lite protocol and the 128-bit AXI4 protocol at the shell (static) side. The decision for 128-bit is based on the size available in Zynq UltraScale+ FPGAs

that provide this size between DDR memory controller and the FPGA fabric. Depending on the exact physical interface required by a module, we instantiate a module wrapper with a set of *bus adaptors* such that a module can communicate with the rest of the system as required by the individual FPGA modules. Figure 4.6 shows a *bus adaptor* being used to translate between different AXI bus standards. We can perform this wrapping at design time (where we instantiate the wrapper transparently when designing the module) or at runtime with partial reconfiguration (where the bus adaptor is a partially reconfigurable module located between the static system infrastructure and the accelerator module). The bus adaptor uses mostly IP components provided by the FPGA vendor Xilinx [32]. These components are then automatically parameterised and integrated according to the specific interface requirements of the accelerator module. With this, shells can remain light-weight, operational, and unchanged while supporting a wide range of AXI interfaces for accelerator modules.

It is important to understand that a static acceleration system would also use such bus adaptors provided by the vendor. Hence, our bus adaptors do not necessarily cause additional overhead unless changing it at runtime which requires pre-allocation of resources. The advantage of the here proposed *bus adaptor* concept is that an adaptor is only integrated into a module if needed and not speculatively provided by the shell.

We also use a bus adaptor to translate between AXI Master and AXI Stream protocols. FOS provides different versions of AXI Stream adaptors to be used depending on the AXI Stream channel width. A user can either re-compile their modules with a logical wrapper of the bus adaptor at design time or stitch their modules with a pre-built partial configuration bitstream binary of that bus adaptor at runtime. Figure 4.7 shows the implementation of the bus adaptor with its logic for interfacing an accelerator with the AXI protocol, AXI MM2S, and AXI DMA services for a module which has a 32-bit AXI-Lite and 32-bit AXI Stream interface.

Module Compilation

The module design begins from either high-level language (HLL) source code (e.g., C, C++, and OpenCL) or a hardware description (RTL/netlist). In the case of HLL source code, we go through a High-Level Synthesis (HLS) step [32] to generate the RTL source code. We then synthesise the resulting RTL source code in out-of-context (OOC) mode in Vivado. In the Xilinx Vivado tools, OOC allows implementing our accelerator modules bottom-up meaning that no optimisations are performed across module boundaries, which is not applicable to partially reconfigurable modules.

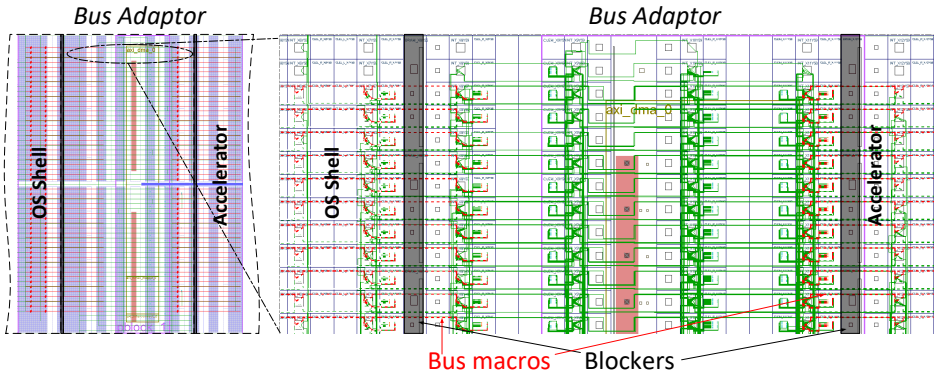
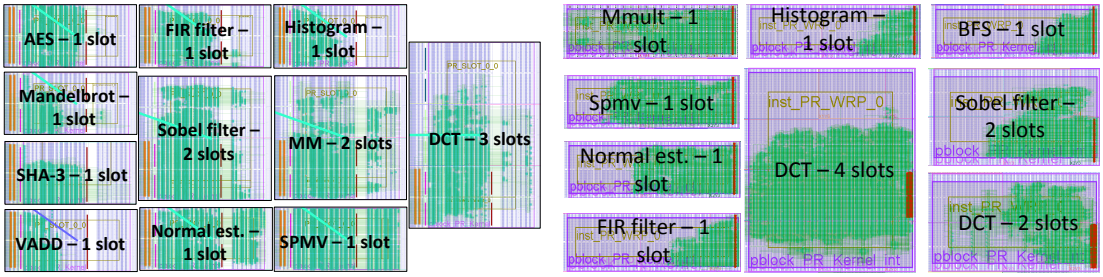


Figure 4.7: Implementation of a bus abstraction layer on UltraZed/Ultra96 platforms. The bus adaptor is provided as an implemented module bitstream and stitched to into the shell at runtime by using partial reconfiguration. The adaptor is a partial module that, in turn, interfaces to other partial modules. This technique avoids re-compiling bus adaptors but comprises an area overhead for a partial region to host a bus adaptor.

The PR Wrapper templates are used to create a minimal top-level placeholder for the physical module implementation. This temporary placeholder acts as sink/source connection points and substitutes the surrounding static system. We then integrate the module OOC netlist into this placeholder for the synthesis and placement stages.

We generate blockers (as TCL routing constraints) to enforce that all partial module's primitives and routing resources are following the strict implementation rules mentioned in Section 4.2.2 (by using GoAhead [6] or TedTCL library [111]). Opposed to the static system where blockers are placed inside reconfigurable regions, here the blockers are placed around the module as a fence for implementing hard module bounding box constraints. The blockers include routing tunnels for the communication to and from the temporary placeholder. The position of these tunnels matches exactly the tunnels used in the static design to implement the communication between static and partial areas. The placeholders and blockers stay the same for all modules requiring the same number of resource slots. For convenience, pre-implemented templates (one for each number of possible slots) are provided in FOS distribution for rapid implementation of accelerator modules.

As we implement a module in separation from the static system, the result generated by Vivado is a full configuration bitstream. We pass this full bitstream to BitMan [84] to extract the configuration data that corresponds to the module only as a partial bitstream. At runtime, BitMan manipulates those partial bitstreams to relocate modules to the desired partial region of the static system. Figure 4.8a and 4.8b shows the resulting modules of the Spector benchmark suite [34] and additional accelerators



(a) (b)

Figure 4.8: Compiled modules for Spector benchmark suite [34] and our in-house accelerators for Ultra-96 and ZCU102 boards in (a) and (b), respectively.

for Ultra-96 and ZCU102 boards, respectively.

4.2.3 Logical Hardware Abstraction

This is the first layer between the hardware and software infrastructure. It is designed to hide the differences or changes in the hardware from the software, in order to detach the software infrastructure from the underlying hardware layer as much as possible. To achieve this conveniently, we propose describing the shell in terms of logical functionalities using a JSON file description providing the following information (see Listing 4.1 for an example):

1. Name of the shell
2. Name of the shell bitstream
3. Partial region:
   (a) Name of the partial region
   (b) Blanking bitstream for the partial region
   (c) AXI bridge decoupler address
   (d) Base address of an accelerator placed in the region

Similarly, for the accelerator, we defined a JSON file description (see Listing 4.2) with the following details for accelerator programming and management:

1. Name of the accelerator
2. Bitstreams:
   (a) Name of the bitstream
   (b) Name of the shell it is compiled for

Listing 4.1: JSON description example of a shell.

```
{
  "name": "Ultra96",
  "bitfile": "Ultra96.bin",
  "regions": [
    {"name": "pr0", "blank": "Blanking_slot_0.bin",
     "bridge": "0xa0010000", "addr": "0xa0000000"},
    {"name": "pr1", "blank": "Blanking_slot_1.bin",
     "bridge": "0xa0020000", "addr": "0xa0001000"},
    {"name": "pr2", "blank": "Blanking_slot_2.bin",
     "bridge": "0xa0030000", "addr": "0xa0002000"}
  ]
}
```

   (c) Type of AXI interface used (default: 128-bit AXI4 master and 32-bit slave)

   (d) Name and number of the PR regions it is compiled for

3. Register mappings:

   (a) Symbolic name of the HW register

   (b) Address offset at which the HW register can be access

Given that the control register map follows the standard Vivado HLS [32] interface (see Listing 4.3), an accelerator can have an arbitrary number of 32-bit registers. This allows building generic drivers for accelerators to relieve hardware developers from the responsibility of writing and integrating drivers. Hence, with this logical hardware abstraction, shells or accelerators can be arbitrarily updated with no need to recompile the Linux kernel or drivers. Section 4.2.4 will describe the driver and acceleration library interface.

The name of the PR region for each bitstream allows backward compatibility to the Xilinx PR flow, where we must compile an accelerator for each region (no relocation support). Moreover, the name of the accelerator is unique, but it may contain bitstreams of varying sizes corresponding to differently sized acceleration implementations providing the same functionality. The scheduler can later use these implementation alternatives to perform resource-elastic scheduling, i.e. dynamically changing the resource allocation used by the accelerator based on the workload and available resources at runtime.

We then register these JSON descriptions for shell and accelerators into a JSON

Listing 4.2: JSON description example for a vector add accelerator.

```
{
  "name": "vadd",
  "bitfiles": [
    {"name": "vadd.bin", "shell": "Ultra96", "region": ["pr0",
        "pr1"]},
  ],
  "registers": [
    {"name": "control", "offset": "0"},
    {"name": "a_op", "offset": "0x10"},
    {"name": "b_op", "offset": "0x18"},
    {"name": "c_out", "offset": "0x20"},
  ]
}
```

Listing 4.3: Control bits for the accelerator.

```
// 0x00 : Control signals
//        bit 0 - ap_start (Read/Write/COH)
//        bit 1 - ap_done (Read/COR)
//        bit 2 - ap_idle (Read)
//        bit 3 - ap_ready (Read)
//        bit 7 - auto_restart (Read/Write)
//        others - reserved
```

based registry to enable a centralised view of the available hardware to the upper software layers. This allows the application developers or the runtime system to request hardware based on just a logical name (a logical accelerator functionality) and corresponding input data, without needing any further information about the underlying hardware layer and accelerator implementation.

Note that we can automatically generate the JSON descriptor for accelerators from the files generated by the Vivado HLS compilation flow [32]. Whereas, the shell developer must write the JSON description to allow the runtime system to manage the resource allocation and use generic drivers. However, this process is commonly required only once per system and can be omitted when using the pre-built shells provided with FOS.

4.2.4 Acceleration Interface Libraries

In general, an OS has to provide high-level APIs that can be used from existing software stacks to improve the accessibility of FPGAs to software developers (who are often non-FPGA experts). One such effort is the PYNQ [127] framework from Xilinx, which allows accessing FPGA accelerators through a high-level Python API. However, the current implementation of PYNQ relies on an existing vendor development flow and contains many direct dependencies on artefacts produced by the Vivado compilation flow which restrict modular development [32, 127]. For instance, it requires the *entire block diagram* of the shell to enable programming some vendor IPs, and this creates a strong direct dependency between shell and runtime environment. While it is possible to engineer around these shortcomings, the resulting system would suffer from code inflation and would be non-scalable due to its legacy support. Hence, we built a new light-weight acceleration interface libraries called Ponq and Cynq for Python and C++ languages, respectively. These libraries are built based on a modular development flow and provide 1) access to static and dynamic FPGA accelerators via its generic drivers for programming accelerators, 2) the Xilinx FPGA manager for partial reconfiguration [128], 3) memory-mapped I/O (MMIO) modules for direct access to accelerators, and 4) a data manager for i) contiguous physical memory allocation and ii) virtual memory protection based on the system MMU [85] which is available on-chip and provides memory encapsulation for the accelerators. For providing extra hardware security against side channel or power hammering attacks, accelerator configuration bitstreams can be checked with FPGADEFENDER [72]. Figure 4.9 shows the integration of Ponq and Cynq libraries into the FOS modular development flow and existing high-level software libraries.

Moreover, the libraries are backwards compatible with the standard Xilinx development flow, i.e. both the PR flow and the static acceleration environment, as we built them on top of the logical hardware abstraction layer. The libraries provide the following basic HAL functionality and generic drivers for hardware acceleration:

- Load an FPGA shell
- Load a partially reconfigurable accelerator
- Load a static accelerator
- Load and program an accelerator based on a logical function name
- Program an accelerator for execution via generic drivers

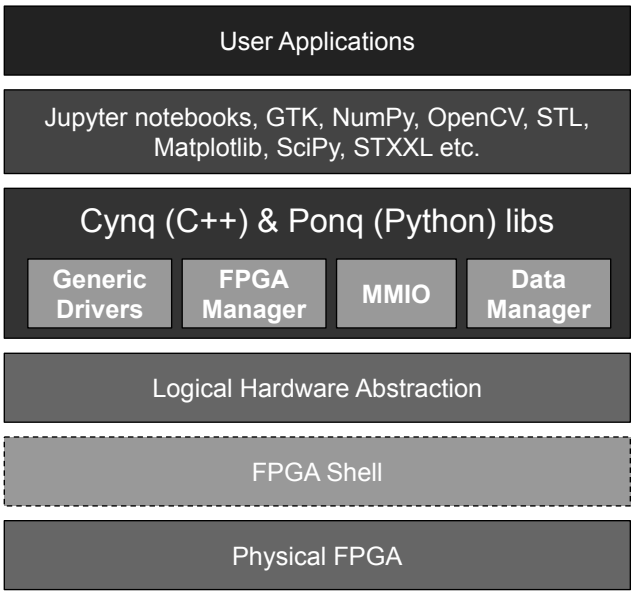


Figure 4.9: Cynq and Ponq libraries as an acceleration interface layer for static and dynamic acceleration on FPGAs.

- Read and write calls to HW registers of the accelerators
- Contiguous physical memory allocation

These functionalities can coexist with other legacy drivers for both the shell or accelerator modules.

4.2.5 Runtime and Multi-tenancy API

To support multi-tenancy, a runtime system is necessary to arbitrate access to reconfigurable resources between multiple users *transparently*. The conventional approach for allocation is to employ time-domain multiplexing with a run-to-completion model, but there has also been spatial domain scheduling mechanism proposed for FPGAs [5]. This approach bases the spatial domain scheduler on pipeline replication. However, it only support a DSL based accelerators and cannot change the resource allocation once it instantiates the modules until the application execution is entirely complete. An operating system, in contrast, must support all types of accelerators and allow them to execute concurrently while using space-time domain scheduling. To make this possible, we need three main components: an API to a daemon scheduler, a programming model and an actual scheduler, as discussed in the following paragraphs.

Listing 4.4: C++ daemon execution call example.

```
1  // Create a job
2  Job &job = jobs.emplace_back();
3  job.accname = "Partial_accel_vadd";
4
5  // Set accelerator parameters
6  job.params["a_op"] = a_op_buffer_ptr;
7  job.params["b_op"] = b_op_buffer_ptr;
8  job.params["c_out"] = c_out_buffer_ptr;
9
10 // Launch jobs
11 fpgaRpc.Run(job);
```

Daemon Scheduler API

To truly support multi-tenancy with portability, we need to design an API which can span across multiple languages and which is portable to different OS kernels or a base processor system with ease. The two efficient ways to perform this inter-process communication (IPC) are 1) message passing and 2) shared memory. In our platform, we adopt the gRPC framework [35], which is a standard RPC framework with support for multi-languages. We use gRPC to send the acceleration requests from the client process to the daemon process, whereas, we pass the data via shared memory to avoid additional latency of copying data (i.e. zero-copy operation). The adoption of gRPC allows us to extend the runtime to accept acceleration requests from remote nodes in the future. The final interface exposed to the application developer, in C++ and Python, is shown in the examples Listing 4.4 and Listing 4.5. Note that each user can offload multiple data-parallel acceleration requests in a single RPC call to the daemon.

Programming Model

Dynamic resource allocation is achieved in the spatial-domain by using resource-elasticity [109] in two forms: module replication and module replacement. However, to make this possible for many types of accelerators, we allow applications to expose data-parallelism to the scheduler, i.e. an application developer can choose to express an acceleration job into a varying degree of parallelism appropriate for the application. A typical example of this is partitioning an image into multiple parts for image-processing accelerators. Note that this is analogous to a software developer deciding

Listing 4.5: Python daemon execution call example.

```
# Create a job and set accelerator parameters
jobs = [{
    "name": "Partial_accel_vadd",
    "params": {
        "a_op": a_op_buffer_ptr,
        "b_op": b_op_buffer_ptr,
        "c_out": c_out_buffer_ptr,
    }]

# set accel parameters and run hardware unit
fpga_rpc.Run(jobs)
```

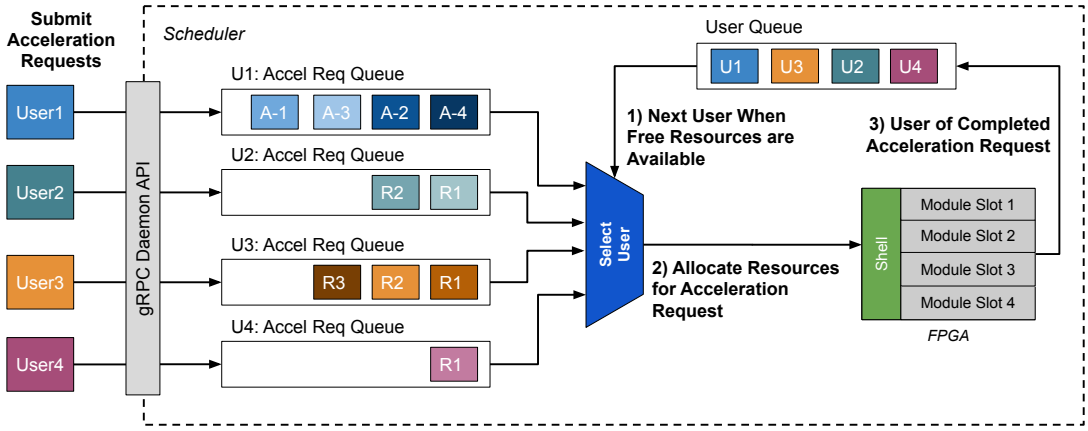


Figure 4.10: Scheduler organisation for resource elastic allocation between different users and their data-parallel acceleration requests.

for the number of threads for efficient execution on a multi-core system. The runtime is then responsible for executing those parts i) in parallel, ii) use a better implementation, or iii) in case of exceeding FPGA resource capacity perform time-domain multiplexing of the resources.

Scheduling

The scheduler maintains a queue for users and performs round-robin scheduling between users at a coarse granularity of data-parallel acceleration requests, as shown in Figure 4.10. Each user also has an individual queue of acceleration requests. Each request in this queue is independent of other requests in the queue and can execute in parallel and any order. At the end of each acceleration request, the scheduler relinquishes the accelerator and selects an acceleration request from the next user in the

queue. This scheme implements a cooperative scheduling policy (by breaking down a job into fine-grained run-to-completion acceleration requests) where each request includes fetching operands and writing back results to main memory (DDR). This corresponds directly to the OpenCL programming model where work-groups can execute in any order. Note that the *scheduling granularity* depends on execution latency of acceleration request rather than fixed timing intervals like in preemptive scheduling.

In the case there is no other user, the scheduler executes requests from the same user in parallel and attempts to use the biggest module (assuming that the biggest module provides the highest performance per resources used, i.e. Pareto-optimal) to maximise the utilisation and performance. Moreover, the scheduler avoids partial reconfiguration and reuses an accelerator if it is already available on-chip. This allows multiple different applications that require acceleration of the same functionality to share the same accelerator in time without paying an additional configuration penalty or user effort.

Consequently, a single task execution call may execute on multiple accelerators in parallel or use an implementation alternative or share an accelerator with other tasks in time, during its execution lifetime. All these modes can be used arbitrarily without an application being aware of other tasks and types of accelerators it executes on. Hence, the runtime system is responsible for dynamically arranging the loading and unloading of these heterogeneous accelerators (written in C, C++, OpenCL or RTL) transparently from the user. Figure 3.6 shows an example of how such resource allocation can allow maximising the FPGA utilisation and performance compared to standard fixed-module scheduling policies.

4.3 Heterogeneous Runtime System

In this section, we describe the implementation of a runtime system to improve the adaptability for heterogeneous FPGA systems by dynamically scheduling multiple applications on both CPU and FPGA.

To achieve this, the resource elastic scheduling problem is solved for heterogeneous resources (CPU and FPGA) in best-effort manner with the help of Branch-and-Bound (BnB) exploration and heuristics. In particular, we design the runtime system to hide the low-level device details with OpenCL interface and transparently enable resource allocation.

The following subsection describes the foundations and components used to build

this runtime system. This is followed by the system overview on how various components in the system integrate with each other as well as the view exposed to the user in Section 4.3.2. Lastly, Section 4.3.3 describes the scheduler implementation for runtime resource allocation.

4.3.1 Runtime Foundations

As mentioned earlier in Section 3.4.2 and 3.5.1, a data parallelism based cooperative approach can be used for treating all the compute resources under a unified execution model. This simplifies the scheduling problem for compute engines with different latencies and resource requirements which provide the same operations.

Given that OpenCL is a widely accepted industry standard for programming data-parallel applications on heterogeneous platforms (i.e. CPUs, GPUs, FPGAs and ASICs), we target OpenCL modelled applications for dynamic runtime scheduling with the extension of sharing all devices transparently. This not only resolves the portability issues across devices but also raises the abstraction to the OpenCL execution model (see Section 3.5.2), as well as hiding the complexity of integrating heterogeneous compute resources (i.e. CPU and FPGA in our case).

To support OpenCL execution on CPU cores and FPGA systems, the POCL runtime [49] and FOS shell (described in Section 4.2) has been used, respectively. The compilation for OpenCL to software is as per the standard LLVM compilation flow (details of which can be found in [49]) and for FPGA accelerators the decoupled compilation flow of FOS was used.

4.3.2 System Overview

Our system runs as a daemon to which multiple different applications can submit kernel execution requests, as shown in Figure 4.11. The runtime contains a waiting queue and a scheduler responsible for executing kernels on FPGA accelerators and submitting sub-kernels (a set of kernel work-groups) to the POCL [49] runtime for CPU execution. However, unlike standard OpenCL runtime systems, we submit work to devices at a fine granularity (work-groups) allowing the devices to be *reallocated* more often and being able of *interleaving* the execution of multiple kernels on the same device as necessary. This approach does not necessarily incur overheads because the native kernel execution is often implemented at work-group granularity internally in runtime systems. For example, for FPGAs, the accelerator is synthesised for the execution of a

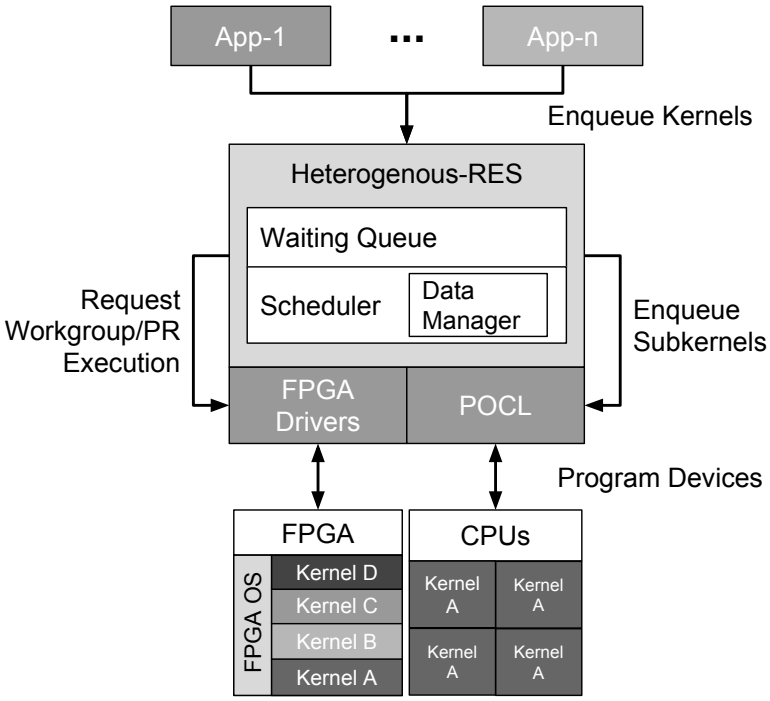


Figure 4.11: The complete execution stack of the runtime system as implemented on the case study platform (ZCU102).

single work-group while for CPUs the kernel is compiled into a work-group function, which is then executed repeatedly for complete kernel execution [55, 56].

At the end of each work-group execution for OpenCL FPGA accelerators, there is no internal state which needs to be stored or I/O in progress, resulting in a natural context-switch point. This allows the runtime to perform context-switching either by directly dispatching a work-group from another application or by replacing the accelerator for a new application using partial reconfiguration (PR). Given the context-switching ability for both device types subjected to cooperative scheduling, our runtime scheduler performs dynamic reallocation of resources for performance optimization and relieve the programmer from the responsibility of *workload partitioning* as well as supporting the *execution of multiple OpenCL applications* across *different devices and accelerator types*.

User view

Figure 4.12 shows the complete design flow used for the development and execution of the OpenCL kernel supporting resource elasticity. The input required by the programmer is an OpenCL kernel with optionally different optimization levels (to generate

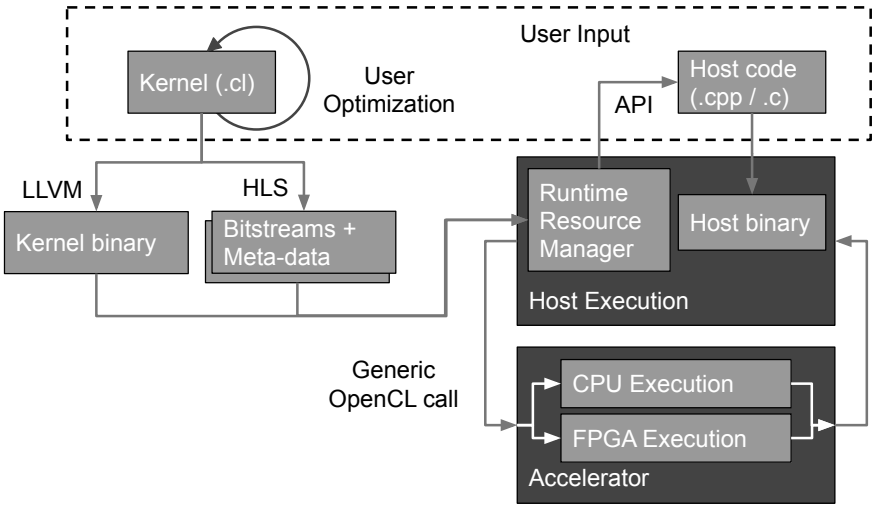


Figure 4.12: Design flow for the OpenCL kernel development (design-time) and execution (runtime) with resource elasticity.

implementation alternatives) and host code. There are two differences compared to the standard OpenCL flow [49,119]: 1) the generation of different kernel versions (e.g., design alternatives with different resource-performance trade-offs) from the user's viewpoint, which can be performed easily by selecting the appropriate HLS pragmas options; 2) a generic OpenCL call is made by the host program rather than device-specific call. This is because the runtime selects the appropriate device type (i.e. to execute the kernel on CPU or FPGA or both) on behalf of the user.

Note that it is not a requirement that each module is implemented with different resource and performance variants as resource elasticity can be implemented by instantiating an accelerator multiple times. The scheduler (see Section 4.3.3) can arbitrarily handle multiple accelerator instances of different size as well as multiple CPUs.

4.3.3 Scheduler

Given that the scheduling problem is a mixed-integer nonlinear problem (see Section A.3), we use a Branch-and-Bound method with snapshot heuristic to explore the resource allocations where ranking functions (domain-specific heuristics) guide the resource management. The algorithm consists of three different steps (as shown in Figure 4.13): kernel selection, resource allocation and resource binding. In summary, the algorithm first identifies a set of kernels to schedule together on the available resources. It then explores the potential resource allocations for the kernels while ranking each resource allocation based on a ranking function (e.g., predicted performance, fairness,

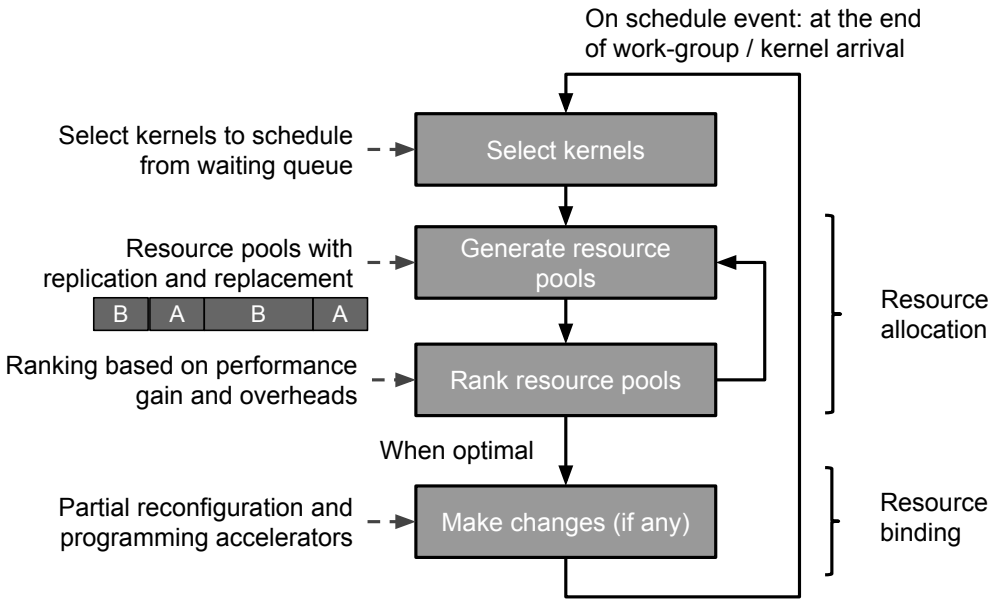


Figure 4.13: Structural overview of our heterogeneous resource elastic scheduler.

energy, QoS or throughput requirements) with consideration of overhead caused by the new resource allocation. The overhead can include the cost of PR, data movement latency and software overhead. After that, once the best possible resource allocation is identified based on a ranking function, it performs the steps necessary to change current resource allocation by performing PR (if needed) and programming of the devices (e.g., FPGA accelerators and CPUs) for kernel execution, as detailed below.

Kernel Selection

To keep waiting times for tasks minimal and ensure kernels continue to maintain progress, we select the maximum number of kernels from the waiting queue. This allows the system to degrade the performance gracefully while over-committing resources. To implement this heuristic, we first relinquish control of as many FPGA accelerators and CPUs as possible (if any) and add them to the back of the waiting queue to free up resources for waiting kernels. At this stage, there may be kernels which have not reached their context-switch points (i.e. currently executing). Hence, we select these kernels in-flight in addition to the maximum number of kernels that we can assign from the waiting queue using the newly recovered resources.

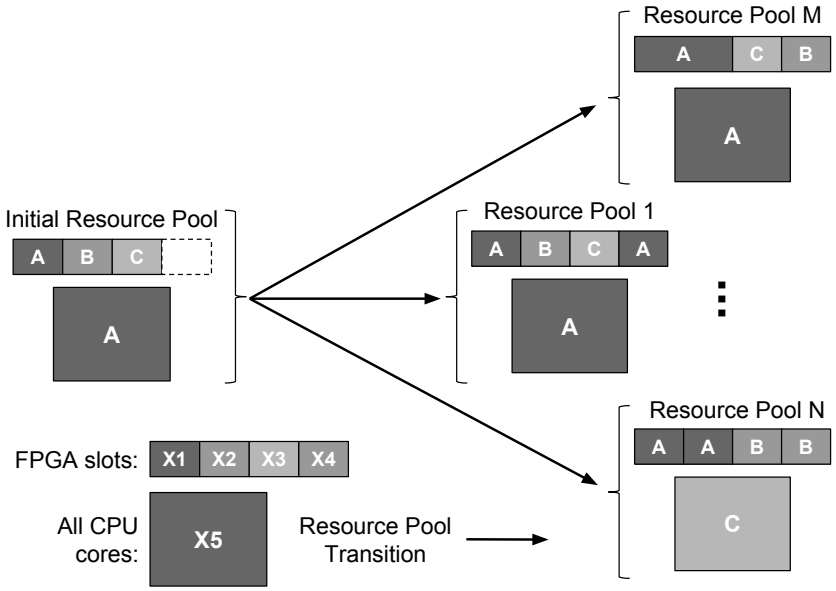


Figure 4.14: Resource pool allocation transition examples upon exit of the kernel hosted in the last FPGA slot (PR region).

Resource Allocation

Given a set of independent kernels (K) to execute, we first identify the current resource pool allocation (c), see Figure 4.14. A resource pool allocation is a collection of allocated resources from the heterogeneous system, i.e. the number of slots (PR regions) on FPGA and CPU cores (see Figure 4.14). Then we perform exploration of the potential resource pool allocation transitions from i to new resource allocation configurations using a Branch-and-Bound algorithm. Equation (4.2) models the makespan of kernels based on a combined rate of execution for CPUs and FPGA accelerators and act as an objective function (f) of the algorithm. Where $T_c(r_k)$ denotes the projected time taken to execute each of the remaining work-items ($W_l(k)$) based on a resource allocation for kernel r_k in the resource pool allocation r given work-item latency $T_w^F(r_k)$ for FPGA allocation and $T_w^C(r_k)$ for CPU allocation, respectively. The transition cost of the resource pool allocation c to r is captured by $O_i(c,r)$, which refers only to the PR cost of the kernel in our system because CPU cores and FPGA slots share the same memory in our target platform (ZCU102), hence removing the need for data movement between private device memory and system memory (which is commonly required when using GPUs or a datacenter FPGA card).

$$T_c(r_k) = \frac{T_w^F(r_k) \times T_w^C(r_k)}{T_w^F(r_k) + T_w^C(r_k)} \times W_l(k) \tag{4.1}$$

$$\arg\max f(r) = \{T_c(r_k) + O_i(c,r) \mid i \in K\} \tag{4.2}$$

The upper bound (g) is calculated using Equation (4.3) where $f(r)$ denotes the execution latency of currently allocated resources, function $T^m(K_l)$ denotes the maximum latency of kernels yet to be assigned to resources (K_l) given their worst-case latency ($T_m(k)$), the number of free FPGA slots s and PR latency per slot P_l, as per Equation (4.4). Note, as we use the worst-case latency of kernels in addition to the worst-case overhead, we achieve an upper bound on makespan latency for a resource pool allocation r. Execution latencies required for these calculations are based on the profiling information provided with the kernel implementation alternatives. Hence, they only serve as a heuristic for problem-solving and are not meant to be absolute. In particular, these heuristics allow estimating the solution of sub-problems and help to reduce the execution time of the algorithm while keeping deviation from the optimal solution minimal by using profiling information.

$$g(r) = max(f(r), T^m(K_l)) \tag{4.3}$$

$$\arg\max T^m(K_l) = \{T_m(k) + s \times P_l \mid k \in K_l\} \tag{4.4}$$

The solver generates decision branches starting with the FPGA whereby a branch for each implementation alternative of the requested accelerators available is created to the first fit location (i.e. starting by branching b number of times where b is the total number of bitstreams of a given kernel K). Intuitively, this can be imagined as filling up the FPGA from one end to the other end one step at a time at each level of exploration tree (see Figure 4.15). Once the FPGA cannot accommodate new modules, our algorithm allocates CPUs and branches with a factor of $|K|$, i.e. CPU allocation for each kernel. However, the total number of branches are subjected to three primary constraints which help in reducing the search space along with the upper bound for recomputing a schedule (g):

1. **Availability**: when only certain sized FPGA accelerators are available rather than all implementation alternatives (e.g., a two-slot FPGA accelerator is infeasible to place when only one slot is free on an FPGA).

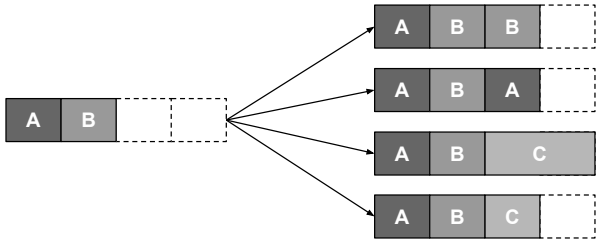


Figure 4.15: Branching cases for our FPGA Branch and Bound allocation algorithm.

2. **Runtime constraints**: executing kernels cannot relocate, which may fragment available free resources.
3. **Over allocation**: we cannot allocate more resources than our heterogeneous system can provide.

Resource Binding

Given a new resource allocation, we identify the necessary changes to the FPGA allocation and queue the request to perform PR through FPGA drivers. Then we program the CPUs by launching sub-kernels via the POCL runtime and the FPGA accelerators via our FPGA drivers. The data manager keeps track of the next work-group, which needs to be scheduled and collects the data back from the accelerators and POCL. Note that our system performs dynamic scheduling of work-groups across different devices rather than static partitioning of kernel workload (work-groups). This reduces the dependence on the accurate profiling information about execution latency as a faster execution unit will automatically receive a larger share of the workload based on the input data characteristics, thus improving performance without user intervention.

4.4 Multi-node Architecture for OpenCL Accelerator Migration

To enable better support for adaptability at the cluster level, in this section, we describe the implementation details of the infrastructure required for live migration methods for OpenCL accelerators (mentioned in Section 3.6) with FOS.

We propose using a distributed system based on a master and slave approach, which is similar to Apache Hadoop YARN [110] (which is one of the industry-standard cluster management systems). The system architecture is shown in Figure 4.16. Here each

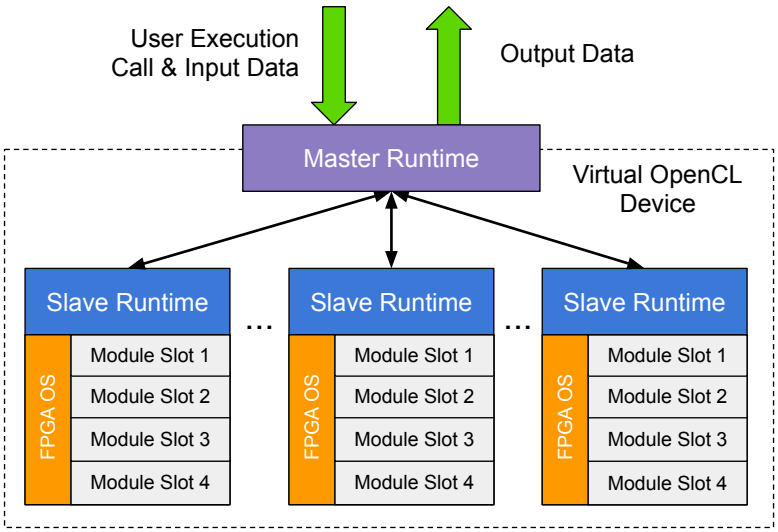


Figure 4.16: Virtualization architecture to abstract number of nodes in the system for OpenCL acceleration.

node has a copy of both master and slave runtime such that the master can be selected based on a standard election algorithm in case of a fault occurrence. The slave runtime system employs resource elastic scheduling such that the number of instances or module implementations can be changed dynamically based on the workload. Further, the slave runtime provides heart-beat signals and checkpoint information to a master node regularly to detect and mitigate unexpected crashes. The master runtime is responsible for host code execution and can distribute workload across multiple FPGAs.

The slave nodes transfer the data payload between slave nodes using Transmission Control Protocol (TCP) to ensure reliability and error checking of the data during transit. The transferred data is then picked up by the slave runtime running on the target node and is used to set up buffers, control state and the programming of OpenCL kernels via partial reconfiguration and generic drivers of FOS.

For non-blocking migration, we need to perform merging of the output data as we execute on both source and target node concurrently (see Section 3.6), this requires identification of the memory locations in buffers written by the source and target nodes respectively. In case of mapping the output data from a source buffer to a target buffer, only the memory locations written by the source must be updated into the target buffer. This can be performed by tracking the memory location with write flags and updating only the memory location in the target buffer where it is set in the corresponding source buffer. However, in most cases, this can be easily determined based on the work-item ID for OpenCL applications as it often corresponds one-to-one or one-to-N with

the index in the output buffer. This allows eliminating the tracking entirely if the information of how many output values each work-item generates. This can either be provided by the user as an additional argument in the execution API call (extension of OpenCL API) or by a compiler as meta-data. In this thesis, we take the API extension approach to minimise the data merging time as current OpenCL compilers for FPGAs do not provide such information. Note, this does not require extra effort from the developer to identify this information as it is a part of the algorithmic design (i.e. how much work each thread needs to perform).

4.5 Chapter Summary

In this chapter, we described the implementation details of 1) hardware and software infrastructure of a modular FPGA OS called FOS, 2) heterogeneous runtime system for OpenCL applications to dynamically scale resources on both CPU and FPGA transparently from the user, and 3) multi-node architecture to off-load work to OpenCL accelerators with live migration support.

The resulting system provides an application-centric view for hardware acceleration to the developers by hiding the complexity encountered when using a heterogeneous CPU+FPGA acceleration with Linux back-end. Moreover, it builds the backbone of an FPGA execution stack at the node level and can be used to support multi-tenancy in a variety of environments with integration to existing virtualisation techniques as summarised in Figure 4.17.

Overall, in this chapter, we implemented the concepts introduced in Chapter 3 to cater to the need of upcoming FPGA systems and allow them to be more maintainable, adaptable and accessible, benefiting both FPGA and application domain experts. Evaluation of performance and overhead of the implemented system is provided in Chapter 5.

Figure 4.17: The software stack and the FPGA shell of FOS can be used to perform resource elastic scheduling (RES) in multiple different OS virtualisation setups due to its light-weight nature and built-in portability. Note that in the figure RES implies both software and hardware infrastructure described in Chapter 4.

Chapter 5

Evaluating Modularity and Resource Elasticity

5.1 Chapter Overview

In this chapter, we aim to evaluate the system components and answer the following questions:

- What is the overhead of FOS in terms of reconfigurable resources, compilation latency, execution latencies of intermediate software layers, memory throughput available to the accelerators, and application performance?
- What is the impact of resource elastic scheduling on performance and utilisation under varying workloads, and how does it compare with standard scheduling policies?
- What is the overhead of performing live migration in a blocking and non-blocking manner for common use cases?

To perform this evaluation, we use a combination of 1) hardware accelerator benchmarks, 2) behavioural simulation, and 3) a memory performance toolkit [71] depending on the experiment setups. The details of the hardware accelerators are provided in the following subsection as they are frequently used in various experiments, while we cover the details of simulation setup and memory performance toolkit as required in Section 5.3.2 and 5.2.3, respectively.

Table 5.1: Spector benchmark suite implementation and its resource usage. Note, we merged two kernels of BFS* by manual in-lining for better overhead representation of the entire application. The resulting bitstream of these accelerators are shown in Figure 4.8b and 4.8a. Resource per slot for each shell can be found in Table 5.2.

Applications	LUTs	BRAMs	DSPs	ZCU102 Slots	Ultra-96 Slots
3D Normal Estimation of Point Cloud	11955	7	39	1	1
Sparse Matrix-Vector Multiplication	14578	2	80	1	2
Sobel Filter	17592	33	8	2	2
Time-domain FIR	7436	6	20	1	1
Discrete Cosine Transform (small)	13671	109	69	2	3
Discrete Cosine Transform (large)	44732	172	341	4	-
Histogram's main kernel	5058	4	3	1	1
Matrix Multiplication	11693	84	50	1	2
Breadth First Search (BFS)*	7204	3	0	1	1

5.1.1 Hardware Accelerator Benchmarks

For the evaluation in this thesis, we mainly use accelerators from the Spector Benchmark suite [34] representing applications from signal processing, machine learning, computer vision, and statistics. This diversity of applications is vital to capture the various compute to data ratios required, as we expect the scheduling workloads and migration overhead to be highly sensitive to it. Further, Spector benchmarks have annotations for various optimisation parameters to test design space exploration tools, which we use to generate different implementation alternatives required to exercise resource elastic scheduling. However, Spector benchmark suite originally targets Altera OpenCL platforms. Hence, we compiled 8 out of 9 application benchmarks (as shown in Table 5.1) to Xilinx platforms, with only the omission of SIMD optimisation pragma which is unavailable for Vivado HLS [124]. We do not consider the remaining application benchmark (merge sort) as it generates incorrect results (at run-time) when synthesised with Vivado HLS[1]. It can be re-written to produce the correct result. However, this would require changing the benchmark implementation considerably.

Additionally, we also use accelerators from Xilinx benchmarks [132], financial models by Ma et al. [70] and our in-house accelerators to enrich the set of accelerator behaviour. The experiment setups identify their details individually when used.

Note that the use of these third-party accelerator modules from different sources

[1]Benchmark suite authors are aware of this issue [34]. However, their proposed workaround is not compatible with Xilinx platforms.

demonstrate that they can be compiled and used without further modifications for systems proposed in this thesis.

5.2 FOS Core Functionalities

There are five primary dimensions onto which we can evaluate an FPGA operating system: 1) FPGA resource overhead, 2) software stack overhead, 3) available memory performance, 4) level of modularity and 5) application performance. We detail and discuss the performance of FOS on each dimension for the Ultra-96 and ZCU102 boards in following subsections.

5.2.1 FPGA Resource Overhead

FPGA shell

The resources used by an FPGA shell have a direct impact on the FPGA resources available for user hardware accelerators. Hence, it is crucial to minimise the overhead as much as possible. Table 5.2 shows the resources available for hardware acceleration when using FOS on ZCU102, Ultra-96, and UltraZed boards. For the ZCU102 board (an MPSoC development kit from the FPGA vendor Xilinx) around 50% of the resources are available for user acceleration whereas on a small IoT category Ultra-96 board it is about 80%. This is because the layout of the ZCU102's chip (XCZU9EG) is irregular, limiting the available resources when supporting relocatable modules (see Figure 4.5). With a regular resource layout like in Ultra-96's chip (a XCZU3EG, see Figure 4.4), we can maximise the resource allocation for relocatable modules and considerably reduce the resource overhead of a FOS shell. However, it is important to note that the resources of the shell are not entirely wasted. They are available for future extensions as well as to implement other host system components, for example, static accelerators or I/O functionalities as done in ECOSCALE [74].

Bus Virtualization

Bus virtualisation is vital for achieving modularity at the hardware interface layer. However, dynamically loading the interconnect wrappers (see Section 4.2.2) requires the pre-allocation of a partial region. Table 5.3 shows the overhead of this pre-allocation

Table 5.2: Available resources for acceleration on the ZCU102 platform and the Ultra-Zed & Ultra96 platforms. The version on ZCU102 has 4 PR regions in total, while the other platforms provide 3 PR regions in total.

Resources on ZCU102 (XCZU9EG)	Number of resources per PR region	Chip utilisation per PR region (%)	Total chip utilisation for accelerators (%)
CLB LUTs	32640	11.70	46.80
CLB Regs.	65280	11.90	47.60
BRAMs	108	12.10	48.40
DSPs	336	13.30	53.20
Resources on Ultra96 & UltraZed (XCZU3EG)	**Number of resources per PR region**	**Chip utilisation per PR region (%)**	**Total chip utilisation for accelerators (%)**
CLB LUTs	17760	25.17	75.51
CLB Regs.	35520	25.17	75.51
BRAMs	60	27.78	83.33
DSPs	96	26.67	80

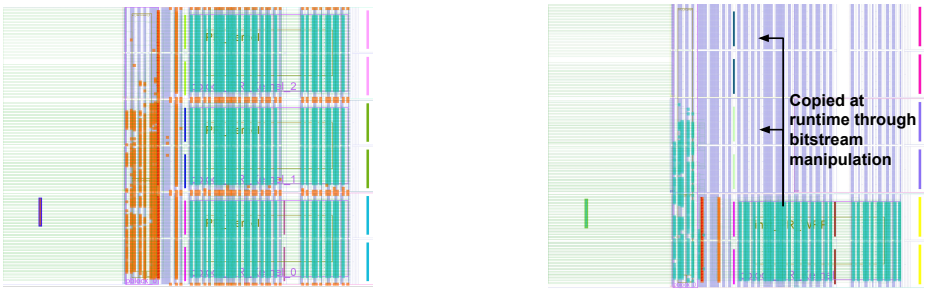
of resources. We can identify that only about 18% (448 LUTs) of the pre-allocated partial region is unused when we change the interconnection interface and protocol considerable, such as from AXI Stream to AXI master and slave. In particular for large FPGAs, this overhead is negligible. However, when using small FPGAs such as on the Ultra-96 or performing minor changes to the interface (e.g., changing bus width), the overhead of dynamic bus virtualisation is considerable (assuming same partial region allocation). Hence, in such scenarios, we recommend using compile-time bus wrappers.

5.2.2 Software Stack Overhead

The software stack of an FPGA OS incurs two types of latencies: compile-time and runtime. Compile-time latency relates to the time taken to compile hardware accelerators with all the additional constraints for partial reconfiguration in the implementation phase. In contrast, the other relates to the overhead caused by intermediate layers in the software stack during accelerator execution.

Table 5.3: Resource overhead of bus virtualisation at logical and physical levels.

Module Interface	Shell Interface	Bus adaptor's services	Resource overhead		
			Primitives	Logical Level	Physical Level
32-bit AXI-Lite & 32-bit AXI4 Master	32-bit AXI-Lite & 128-bit AXI4 Master	AXI Interconnect	LUTs	153	2400
			FFs	284	4800
			BRAMs	0	12
32-bit AXI -Lite & 32-bit AXI Stream	32-bit AXI-Lite & 128-bit AXI4 Master	Control reg., AXI MM2S, & AXI DMA	LUTs	1952	2400
			FFs	2694	4800
			BRAMs	2.5	12



(a) Xilinx PR compilation flow result.

(b) FOS compilation flow result.

Figure 5.1: Place and route results of Black Scholes accelerator [70] for Xilinx PR flow and FOS. Note, FOS uses BitMan [84] to generate relocatable bitstream which is copied to other PR regions.

Compilation Latency

The standard Xilinx partial reconfiguration (PR) flow requires compiling both static and accelerator designs together and, more importantly, it needs to perform place and route (P&R) and the generation of a dedicated bitstream for each partial region. This leads to additional latency compared to our decoupled compilation flow, where we first generate a *full-static* bitstream with Vivado and then a *relocatable* partial bitstream with BitMan [84] for all regions. To quantify this, we used accelerators of three different sizes: sparse (AES), medium (Normal Est. [34]) and dense (Black Scholes [70]). The utilisation for each is 33%, 63% and 81%, respectively. Figure 4.8a shows the AES and Normal Est. modules while Figure 5.1 shows the Black Scholes module with a comparison to a design generated by the Xilinx PR flow.

Table 5.4 shows the latency breakdown for place and route and bitstream generation for both Xilinx PR flow and FOS compilation flow on Ultra-96. The results show

Table 5.4: Total place and route (P&R) times, and bitstream generation latency for AES, Normal Est. [34], and Black Scholes accelerator [70] when compiling for all three partial regions on Ultra-96 shell. The evaluation is conducted using Vivado 2018.2.1 on an Intel core i7-4930K CPU running at 3.4 GHz with 64 GB of RAM.

Applications	Region Util.	Xilinx PR (3x one region)			FOS (one region)			Speed Up
		P&R (s)	Bitgen (s)	Total (s)	P&R (s)	Bitgen (s)	Total (s)	
AES	33%	429.40	176.19	605.59	284.18	64.06	348.24	1.74 ×
Normal Est.	63%	747.75	201.21	948.96	387.41	70.09	457.50	2.07 ×
Black Scholes	81%	1296.26	231.27	1527.53	574.56	77.11	651.67	2.34 ×

that P&R latency per region is higher for FOS as it adds additional constraints for supporting relocatability, however, when compiling for multiple regions (i.e. 3 for Ultra-96 platform) it outperforms the traditional compilation flow (by up to 2.34×) by duplicating a single slot accelerator. Overall, when increasing the number of partial regions, the compilation flow latency of FOS stays constant, whereas the latency of the Xilinx PR flow increases linearly. This relationship turns into an exponential increase in compilation time when compiling several applications with standard Xilinx PR flow, as it needs to compile *each module for each partial region*. Hence, it is necessary to adopt a scalable and modular PR flow when targeting multiple applications and shell versions which are common in datacenter environments.

Runtime Execution Overhead

The runtime overhead occurs because of the four main steps performed when using the FOS software stack: i) initialisation of the gRPC server, ii) parsing of JSON files for accelerators and shells, iii) gRPC calls to the daemon and iv) scheduling latency. Table 5.5 details the latency of each step. The first two steps (i and ii) have dependencies on I/O as they use the network and file system, respectively. This leads to latencies in the range of milliseconds. However, this overhead is amortised over time as we perform steps i) and ii) only once when system starts. The gRPC call to the daemon goes through many levels of the Linux stack (processes) before reaching the FOS multi-tenancy daemon, leading to a latency of about one millisecond. We can speed up the gRPC call by reducing the Linux timer interrupt period to achieve a better response time from the Linux kernel for a quick turnaround between processes if required. The scheduling latency is in the range of microseconds and comparable to standard CPU

Table 5.5: Execution overhead caused by various software layers.

Software Layer	Latency (ms)
Initialize gRPC (once)	12.20
JSON parsing (once)	2.27
gRPC Call to Daemon	0.71
Scheduler	0.02

schedulers. Moreover, the scheduler is event-driven and executed only when an accelerator finishes or a new acceleration request arrives (due to its cooperative nature) rather than at every timer interrupt like preemptive scheduling.

5.2.3 Memory Performance

One central characteristic of an FPGA operating system on the hardware acceleration performance is the memory bandwidth that it can provide given the shell implementation (interconnect to the memory). Hence, we evaluated the available memory throughput of different 128-bit High-Performance AXI ports (HP ports) provided to partial regions on the FOS shells for ZCU102 and Ultra-96 boards with accelerators running at 100 MHz using memory evaluation kit proposed in [71]. In particular, we measured the performance of each port for 1) read only, 2) write only, and 3) aggregated read-write transactions of varying burst sizes from the PL (FPGA). Additionally, we examine the combined performance of all ports running in parallel.

Figure 5.2 shows the breakdown of the read and write throughput of each AXI port and the total accessible bandwidth for the Ultra-96 board, respectively. On average, there is an even split of read and write bandwidth with a throughput of 530 MB/s for each read and write operation. The aggregated read-write throughput of the individual AXI ports is about 1060 MB/s and, when activating all three ports at the same time collectively they achieve up to 3187 MB/s. This translates to about 25% and 74% of the theoretical DDR peak throughput when using AXI ports individually and concurrently, respectively.

Figure 5.3 shows the breakdown of read and write throughput on the ZCU102 board. Each AXI port achieves an even throughput distribution of 1600 MB/s between read and write transactions. The total throughput is 3200 MB/s for individual AXI ports and 8804 MB/s when using all AXIs together, as shown in Figure 5.3. A possible explanation for this sub-linear improvement in the total throughput when using all AXI

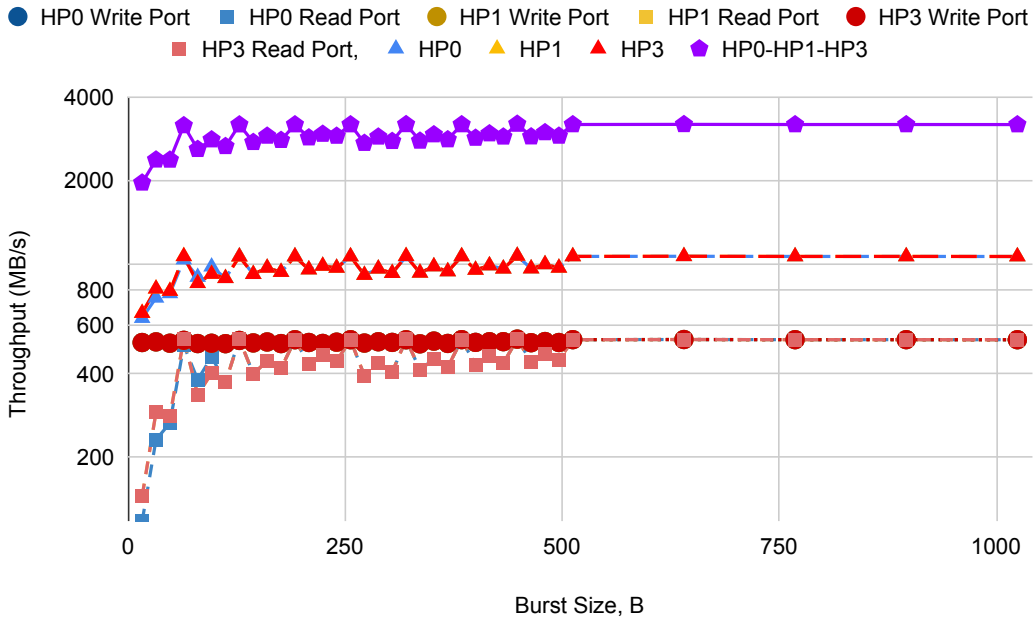


Figure 5.2: Memory throughput for varying burst sizes on the 128-bit duplex High Performance AXI ports (HP0, HP1, and HP3) available to the hardware accelerators in PR regions 0 to 2 of Ultra-96 FOS platform at 100 MHz.

ports (HP0-3) concurrently is because of the row pollution and AXI interconnect multiplexing in the memory controller. This result also reveals that the shell is saturating the memory throughput that is available through the ARM SoC and that it is not worth to spend more resources in the shell to implement wider backplane buses.

5.2.4 Quantifying Modularity

Compared to the standard FPGA development flow, where a change in the shell means recompilation of all system components (both hardware and software) [56, 128], the modular FOS FPGA development flow provides the freedom to *update individual components* without recompilation of other components which may take hours [32,57,128]. This means that FOS only has to pay compilation and re-initialisation latency (as shown in Table 5.6) for the changed component given that it does not modify the component's defined interface.

From Table 5.6, we can identify that this avoidance of recompilation allows changing the shell at runtime with additional functionalities or bug fixes costs less than 21 ms

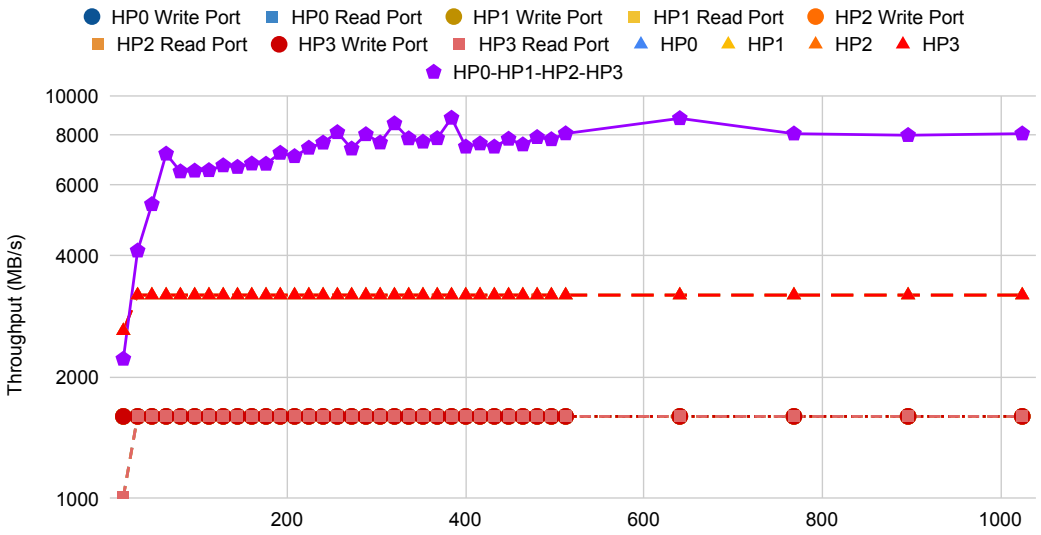


Figure 5.3: Memory throughput for varying burst sizes on the 128-bit duplex High Performance AXI ports (HP0 to HP3) available to the hardware accelerators in PR regions 0 to 3 of ZCU102 FOS platform at 100 MHz.

on Ultra-96 (IoT device) and 99 ms on ZCU102 (Xilinx MPSoC development kit)[2]. Similarly, swapping an accelerator implementation is easy and includes only the partial reconfiguration latency, because FOS provides generic drivers. In contrast to FOS, the standard flow would require generating/writing new drivers and re-installing them separately. Changing the kernel involves the most significant re-initialisation latency, as this includes a system reboot which takes 66 seconds, including I/O setup (for keyboard, mouse, Wi-Fi, monitor and webcam) on Ultra-96[3]. However, this still avoids the need to compile user software binaries for re-integration as required in the standard PetaLinux flow [128], causing latency in tens of minutes (11 and 16 minutes for Sobel [132] and in-house Mandelbrot applications, respectively).

Overall, the modularity of FOS removes the recompilation latencies and redevelopment steps (i.e. mandatory recompilation of all shell, accelerators, Linux and user applications caused by Vivado [124] and PetaLinux [128] for any changes in single component like shell) which cost in the range of hours [57]. This reduces the component change latency by two-orders of magnitude compared to the standard development

[2]This assumes that the shell bitstream is pre-compiled as recompilation of the shell itself for bug fixes or new features is unavoidable for all systems.

[3]The Ultra-96 board support package and vendor drivers included for Wi-Fi and bluetooth have a bug which causes time-out during boot up period, inflating reboot time. After boot-up, our work-around in Ubuntu setup fixes the bug to operate Wi-Fi correctly.

Table 5.6: Re-initialisation latencies for component change on FOS platforms. Note that in contrast to FOS, for any changes in these components, standard PetaLinux flow [128] requires a generation of new Linux image, and hence, a complete reboot and re-initialisation of all components even after excluding compilation latencies.

Component Updated in FOS	U-96 Latency (ms)	ZCU102 Latency (ms)
Accelerator (one slot)	3.81	6.77
Shell	20.74	98.4
Runtime	15.2	15.2
Kernel (including reboot)	66000[3]	15760

flow while supporting all the features required from an FPGA OS.

5.2.5 Application Case Study

We evaluated a case-study in two different environments: 1) single-tenant but multiple partial regions and 2) multi-tenant with dynamic offloading. All the accelerators used in this case study operate at default frequency of ZUCL 2.0 [85] shells (100 MHz) as a baseline environment.

Single-tenant with Multiple Partial Regions

To examine the benefits of multiple partial regions and the ability to replicate accelerators dynamically, we selected OpenCL accelerators from the Spector benchmark suite [34] for reasons mentioned in Section 5.1.1. Figure 5.4 shows the results of the execution latencies with a varying amount of resources available for acceleration on the ZCU102 platform. Most of the benchmark applications show an almost *linear* performance improvement when replicated across multiple partial regions. This is expected from our memory evaluation experiments (in Section 5.2.3) which show that multiple ports can be used in parallel without major performance drops on individual memory port performance. The effect of implementation alternatives can be seen for the DCT accelerator which benefits from the ability to switch to a bigger module implementation (by using larger data buffers and a larger unrolling factor in the bigger alternative) and which achieves a *super-linear* performance improvement of 3.55× for 2× the resources.

To understand the effects of exposing more parallelism than available on the FPGA platform, we conducted the experiments on Ultra-96 using compute-bound (Mandelbrot and Black Scholes [70]) and memory-bound (Sobel [132]) applications which are

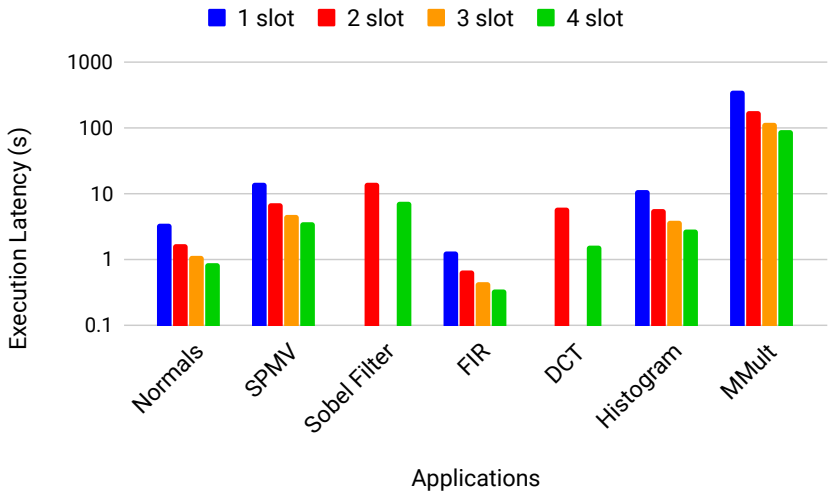


Figure 5.4: Execution latencies of OpenCL accelerators from the Spector benchmark suite running on the ZCU102 platform.

more sensitive to reconfiguration overhead due to their smaller execution latencies than Spector benchmarks [34]. Figure 5.5 shows the results in which the number of requests dictates the amount of parallelism exposed to the runtime system. The performance improves almost linearly until it hits the number of available partial regions (3 in the case of Ultra-96 platform), after which the performance tends to stagnate (see Figure 5.6). This is because the scheduler uses time multiplexing to provide the illusion of an unlimited number of regions, leading to a behaviour similar to multi-threading on CPUs. In particular for cases with the number of requests which are a multiple of the number of physical accelerators, we see better performance than in other cases as they avoids unused resources which are caused by the (leftover) pending requests at the end of the execution of a job queue.

Overall, this highlights that FOS can help to improve performance (by up to 2.54×) even when using a single application along with its other benefits of modularity and developer productivity.

Multi-tenant Dynamic Offload

To understand the performance changes when using multiple applications at the same time, we execute the Mandelbrot and Sobel [132] applications concurrently on the Ultra-96 platform. Note that individual applications are not aware of other applications executing on the platform. In particular, the accelerators that are written and

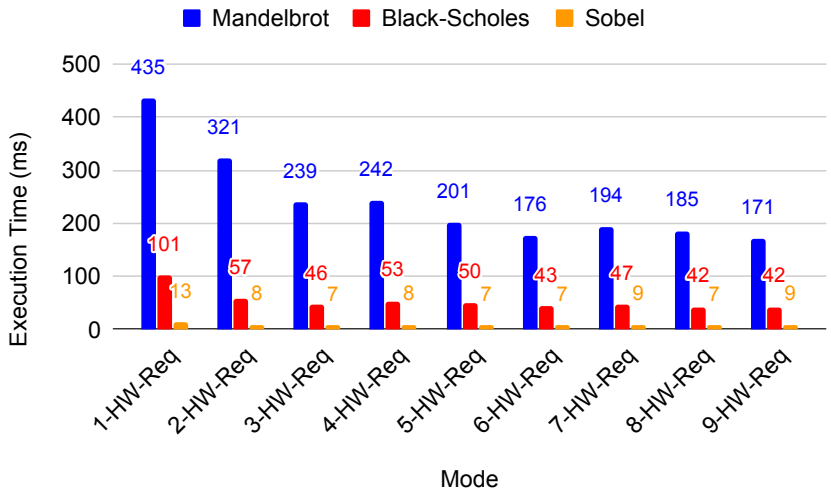


Figure 5.5: Execution latencies of Mandelbrot, Black Scholes (European option) [70], and Sobel [132] when executing concurrently with varying the amount of hardware requests on the Ultra-96 platform.

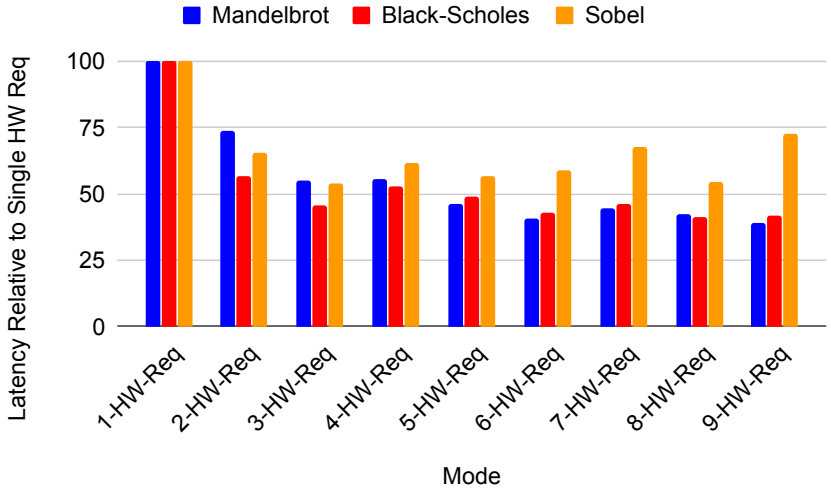


Figure 5.6: Relative execution latencies of Mandelbrot, Black Scholes (European option) [70], and Sobel [132] applications when exposing varying amounts of parallelism to process a frame on the Ultra-96 platform.

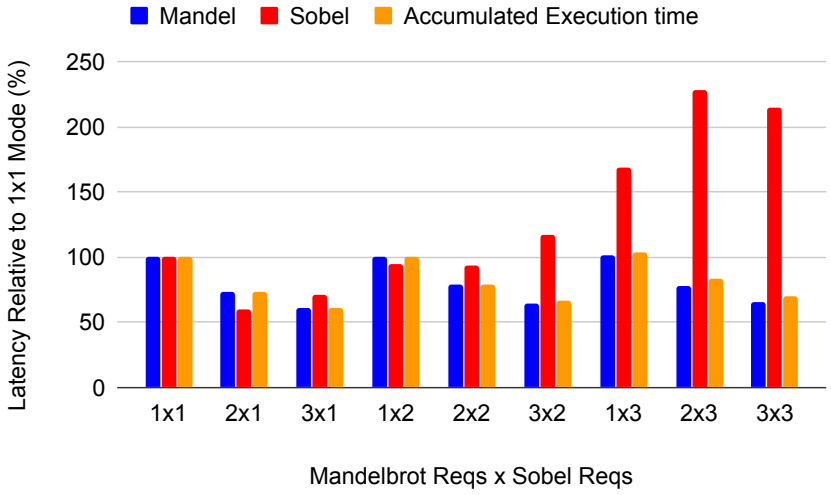


Figure 5.7: Relative execution latencies of Mandelbrot and Sobel [132] when executing concurrently with varying amounts of HW requests on the Ultra-96 platform.

accessed (at runtime) in C (Mandelbrot) and OpenCL (Sobel), demonstrate the support for multiple different languages used *concurrently*.

Figure 5.7 shows the execution latencies relative to 1-Mandel×1-Sobel scenario with a varying amount of acceleration requests. As we can see, the execution latencies tend to decrease with an increase in the number of requests as in the standalone case. However, here the optimal performance is achieved at 3-Mandel×1-Sobel rather than 3-Mandel×3-Sobel. This is because of two reasons: 1) adding more Sobel units reduces memory performance due to row-bank pollution and 2) multiple requests from different applications (over capacity) induces higher reconfiguration overhead for swapping them in and out because of FOS scheduler's fairness policy (see Section 5.3 for performance-oriented scheduler's evaluation). Regardless of this behaviour, it is essential to note that if each application takes a greedy decision to request the highest amount of parallelism suited for the application based on the standalone results, the system can still achieve a near-optimal performance resulting in 46% improvement over 1-Mandel×1-Sobel by dynamically reallocating resources using the same accelerators.

5.2.6 Summary

Overall, the evaluation shows that FOS can speed up the compilation time by up to 2.34× and avoid standard recompilation requirements to reduce update latencies by

over 100× due to its modular FPGA development flow. The overheads caused by the hardware and software layers are minor and recovered by the ability to schedule resources dynamically in both single and multi-tenant environments.

In comparison to BORPH's UNIX interface [99] where hardware process creation can take up to 900 ms, an acceleration request to FOS (i.e. gRPC call to daemon in Table 5.5) takes only 0.71 ms. Moreover, FOS avoids the latencies required for synchronisation (2 ms) and communication message queues (2.16 ms) on top of partial reconfiguration when using thread interface from ReconOS [67].

With this, FOS directly caters the needs of upcoming FPGA systems which are being deployed at scale (cloud and edge) and allow systems to be more maintainable, adaptable and accessible, benefiting both FPGA and application domain experts.

5.3 Heterogeneous Runtime System

The evaluation of the heterogeneous runtime system (described in Section 4.3) is performed in two-stages. First, we undertake a case study with real-world workloads using the Spector benchmark to identify the feasibility and practical performance benefit achievable with resource elasticity on ZCU102 platform of FOS (see Section 5.3.1). Then we carry out behavioural simulation experiments with varying workload sizes, workload types and compute resources to assess the scalability and behaviours of the schedulers operating under worst-case scenarios (see Section 5.3.2).

5.3.1 Application Case Study

For this evaluation, the runtime (see Section 4.3) uses POCL version 1.2 [49] and Ubuntu 16.04.5 LTS based on the Xilinx PetaLinux 2018.2 kernel. We use POCL in standard configuration with automatic vectorisation enabled in LLVM, which maps the work-items operations to NEON vector instructions whenever possible. However, at this point, POCL does not support the separation of compute units as individual devices like other OpenCL runtimes and hence executes the OpenCL kernel on all four cores concurrently[4]. Consequently, we treat the system as a single logical CPU, together with a 4-slot FPGA.

We select four OpenCL accelerators from Spector benchmark suite [34] with different device type preferences, i.e. applications that perform better in software (3D

[4]Note, POCL is the only open-source OpenCL runtime which supports ARM CPUs.

Table 5.7: Relative performance of accelerator benchmarks on various resources.

Benchmark	CPU	1-slot	2-slot	3-slot	4-slot
Normal est.	4x	**1x**	2x	3x	4x
FIR	**1x**	3.4x	6.8x	10x	13.6x
DCT	1.18x	-	**1x**	-	3.54x
MMult	**1x**	3.8x	7.6x	11.4x	15x

Normal estimation (Normal est.)), in hardware (Matrix-Matrix multiplication (MMult) and Finite Impulse Response (FIR) filter) or reasonably good in both cases (Discrete Cosine Transformation (DCT)). Figure 4.8b and Table 5.7 shows the compilation results of these, where Normal est. is CPU favoured and has a 4× performance benefit over its FPGA implementation, FIR is FPGA favoured with 3.4× performance benefit over CPU, DCT on CPU is 1.18× faster as compared to a 2-slot FPGA accelerator and 3× slower than a 4-slot FPGA accelerator and MMult is 3.8× faster on the FPGA compared to CPU only. The implementation difference between 2-slot or 4-slot DCT modules is due to the larger buffer sizes and loop unrolling for better data reuse. In particular, DCT serves as an interesting edge case because, depending on the version which a scheduler considers, it may be CPU favoured or FPGA favoured. Note that we compiled the benchmarks unmodified with default compilation settings and that our approach does not require any changes to the OpenCL code used.

For simplicity, our case study schedule comprises the following arrival times: Normal est. at 0ms, DCT at 100ms, FIR at 200 ms and MMult at 300ms. We consider five main execution scenarios:

1. **Run-to-completion (RC)** execution only on CPUs using POCL.
2. Run-to-completion execution on static FPGA accelerator and CPU (**SDSoC [56] like execution**) where the FPGA accelerator is for longest kernel (MMult).
3. **Round-robin with heuristic (RR-H)** for preferred device type with execution on both CPU and FPGA.
4. **Resource elastic scheduling (RES)** only on FPGA.
5. **Heterogeneous resource elastic scheduling (HRES)** on both CPU and FPGA.

Figure 5.8 shows the resulting waiting time and makespan latencies. Introducing the FPGA accelerator improves the makespan by 4× over CPU only as MMult gets

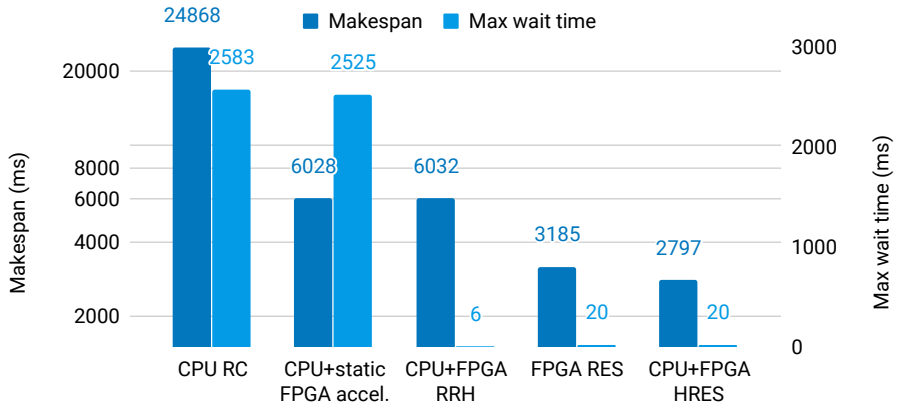


Figure 5.8: Wait times and the total makespan for the HRES case study.

to execute on its preferred execution unit. Further, the CPU+static FPGA accelerator system performs similar to RR-H but results in very long wait times as it obeys the run-to-completion model. Overall, FPGA with RES and HRES achieve speedups of 1.89× and 2.1× with 126× improvement in max waiting time over SDSoC-like systems, respectively. Moreover, compared to RES (FPGA only), its heterogeneous version achieves a 13% better makespan using the same platform and accelerators.

5.3.2 Simulation Experiments

In the previous section, we evaluated the physical implementation of the system with a small set of benchmarks to identify the feasibility and comparative gains over commercially available FPGA platform (SDSoC [56]). However, this does not help us evaluate the scheduler behaviour under varying workload scenarios and FPGA sizes due to its physical limitations and lack of scheduling workload benchmarks. To asses the scalability and behaviour of dynamic schedulers, in this section, we use a *behaviour simulator* (accurate to a millisecond) with varying workload sizes, workload types and compute resources.

Workload Characterization

We tested the system with varying workload level, whereby the compute-bound kernels arrive at the rate of (1, 5, 10, 15, and 20) kernels per second for 100 seconds based on a Poisson distribution, i.e. on average each schedule is made up of (100, 500, 1000, 1500, and 2000) kernels respectively. Additionally, each scheduling scenario is sampled 1000 times for error range and significance.

To model, a heterogeneous workload scenario where certain applications run faster on CPU while others run faster on FPGA, we split kernels into two categories: CPU favoured and FPGA favoured. The ratio of CPU favoured to FPGA favoured is assessed by three variants: 25% CPU - 75% FPGA, 50% CPU - 50% FPGA, and 75% CPU - 25% FPGA. We run these kernels on platforms with CPUs ranging from 0 to 4 and FPGA slots ranging from 0 to 8.

Characteristics of each kernel are generated based on a uniform random distribution, i.e. base latencies of work-groups are in the range of [20, 100] milliseconds, arrival times are set randomly in the Poisson time interval, the slot size of each kernel bitstream and the number of bitstreams is independently in the range of [1, min(4, n)] where n is the number of available FPGA slots. It further receives a speed-up on the base latency in the range of [2, 16] if the bitstream size is greater than one or if the kernel is CPU favoured. To model a wide variety of kernel sizes, the total number of work-groups ranges in [10, 1000]. We assume the I/O requirements of the kernels are not on the critical path of the application. The PR latency is modelled linearly with respect to slot size, where each slot requires on average 6.77 milliseconds for reconfiguration (as found in Section 5.2). Note, this implies that there could be kernels where the PR latency can be as much as $4\times$ the work-group execution latency as well as kernels where PR overhead is negligible. Restricting the maximum number of slots per bitstream allows to investigate how the scheduling behaviour changes when a larger FPGA is available for the same workload (i.e. FPGAs with 6 and 8 slots).

Schedulers

We execute these randomly generated scheduling requirements on five different scheduling algorithms on the *same platforms with both CPU and FPGA*. Four of which serve as a baseline for our experiments:

1. **Run-to-completion (RC)** is the standard for OpenCL runtime systems [56, 119, 126] but with multiple partial regions (PR) for FPGA to support multiple accelerators (e.g. SDSoC [56] with multiple PR regions).
2. **Run-to-completion with heuristic (RC-H)** selects the device based on profiling information (execution latency) whenever both (CPU and FPGA) are available for execution. This is similar to the approaches used in [2, 67, 81].
3. **Round-robin scheduling policy (RR)** is using the context-switching mechanism discussed in Section 4.3, to allow reallocation of PR regions at the end of

work-group to other kernels. This is similar to the methods used in [43, 120] but with an extension to scheduling on both CPU and FPGA.

4. **Round robin with heuristic (RR-H)** couples a standard round-robin policy with a heuristic to select the favoured execution unit when there is a choice between device type (CPU and FPGA). This is similar to approaches used in [7, 20, 37] but with additional support for partial reconfiguration and multi-tasking to share resources among multiple applications.

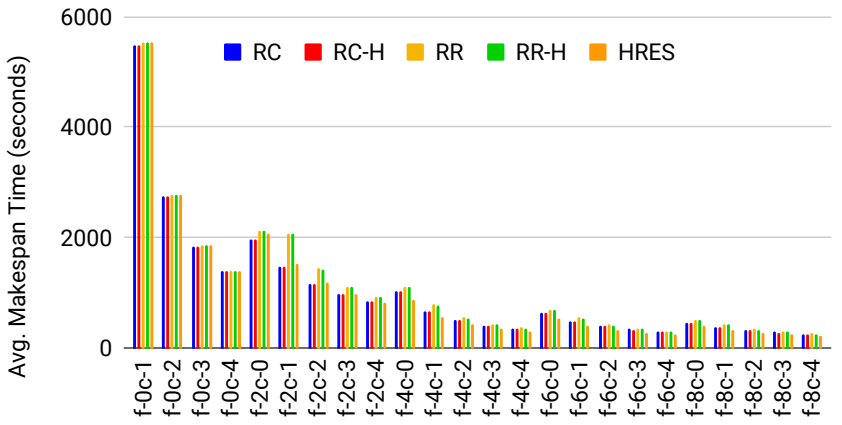
These four baselines are compared against (5) **heterogeneous resource-elastic scheduler (HRES)** which follows the implementation described in Section 4.3.3. Note that the platforms with no CPUs (f-*n*c-0) or no FPGA slots (f-0c-*m*) capture the results of performing resource elastic scheduling for only a single device type, i.e. either FPGA or CPU only. Additionally, in the behaviour simulator, we remove the POCL's implementation restriction of executing on all cores [49] to allow the schedulers to change the number of CPU cores dynamically.

Overall, our simulation experiments only change execution models (as above) and keep the platform capabilities and workload the same for evaluating the impact of schedulers on system performance and overheads.

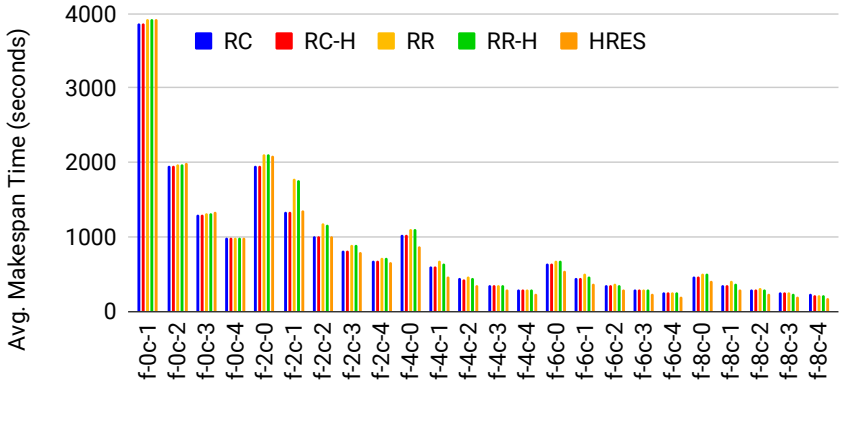
Results

Figure 5.9a, 5.9b, and 5.9c show the makespan (time to completion) latency of all schedulers with kernels arriving at the rate of 5 per second with 25%, 50% and 75% of the kernels being CPU favoured, respectively. We can see that adding more resources to the execution platform allows the kernels to execute in parallel and reduce the makespan as someone would expect. When the skewed workload executes completely on it non-favoured device type (e.g. FPGA favoured kernels on CPU), the execution latency increases substantially. However, adding the smallest instance of the preferred device types helps reducing the latency considerably.

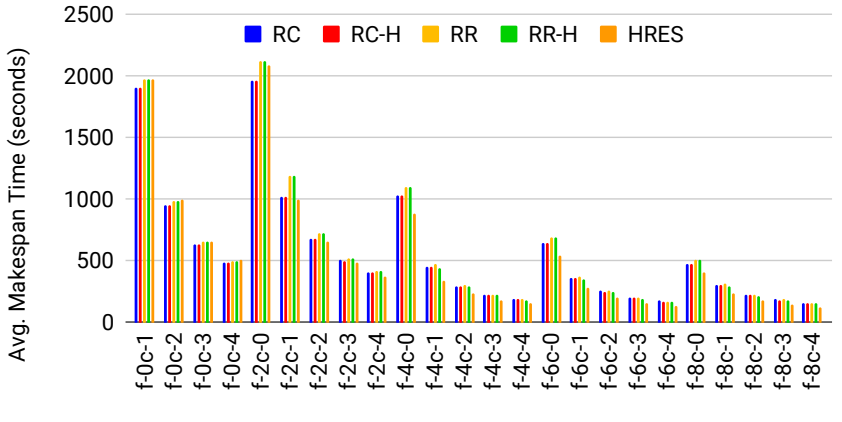
If we look at the relative performance of HRES compared to the run-to-completion scheduler on varying CPU-to-FPGA favour ratios in Figure 5.10a, we can see that overall, when averaged across all workload sizes, the general trend is similar. When the number of resources available is small, HRES does not show any particular benefit and in fact, an overhead of 6% (data point f-2c-0) in the worst case. This quickly turns into 20% performance improvement on average as we add more resources to



(a)



(b)



(c)

Figure 5.9: Makespan time for an arrival rate of 5 and CPU-to-FPGA ratio of 25:75 in (a), 50:50 in (b), and 75:25 in (c) on various execution platforms, where f-*n*c-*m* denotes *n* FPGA slots and *m* CPUs.

the system allowing HRES to allocate free resources dynamically for task acceleration. HRES maintains this benefit of around 15 to 20% improvement even with an increase of 20× in the workload size. This demonstrates that the scaling in workload size does not impact the benefit of HRES over a run-to-completion model. Figure 5.10b, 5.10c, and 5.10d show a detailed breakdown based on kernel arrival rate for varying CPU-to-FPGA favoured ratio. The exact latency behaviour changes minorly based on the ratio of CPU-to-FPGA preference of the workload for kernel arrival rates above 1. The low arrival rate of 1 shows a deviation on how quickly the performance changes with an increase in resources. This is because lower workload allows more opportunities for collaborative execution (i.e. executing a kernel on both CPU and FPGA), hence, improving performance rapidly.

We measure wait time as the difference between the arrival time of a task and the time this task begins execution (including PR latency). We can see from Figure 5.11a that, on average, the wait time is reduced by 95% when using HRES compared to a run-to-completion model. The exceptions to this are pure FPGA platforms and highly CPU favoured workload. Pure FPGA platforms force the scheduler to wait for partial reconfiguration before starting the execution, hence, increasing the wait time. Whereas for CPU+FPGA platforms, launching the kernel on the CPU while reconfiguring the FPGA slot reduces the wait time. The outlier behaviour of kernel arrival rate of 1 mostly skews the wait time for a workload with 75% of kernels being CPU favoured as shown by the breakdown in Figure 5.11d. This is because when the workload is low, HRES takes a greedy decision of performing collaborative execution by allocating all resources to a single kernel, this induces wait times for arriving kernels. In contrast, a run-to-completion scheduler would refrain from allocating more resources than minimally required by the kernel, leaving the resources free for incoming kernels and keeping the wait time near zero. We can also see the effect of this decision on the relative makespan in Figure 5.10d, where makespan latency improves. However, run-to-completion loses this advantage as the workload size increases, i.e. when the number of kernels increases higher than the number of compute units. Overall, HRES's advantage over run-to-completion in waiting time also scales with respect to the workload and size and types as shown in the detailed breakdown of waiting times in Figure 5.11.

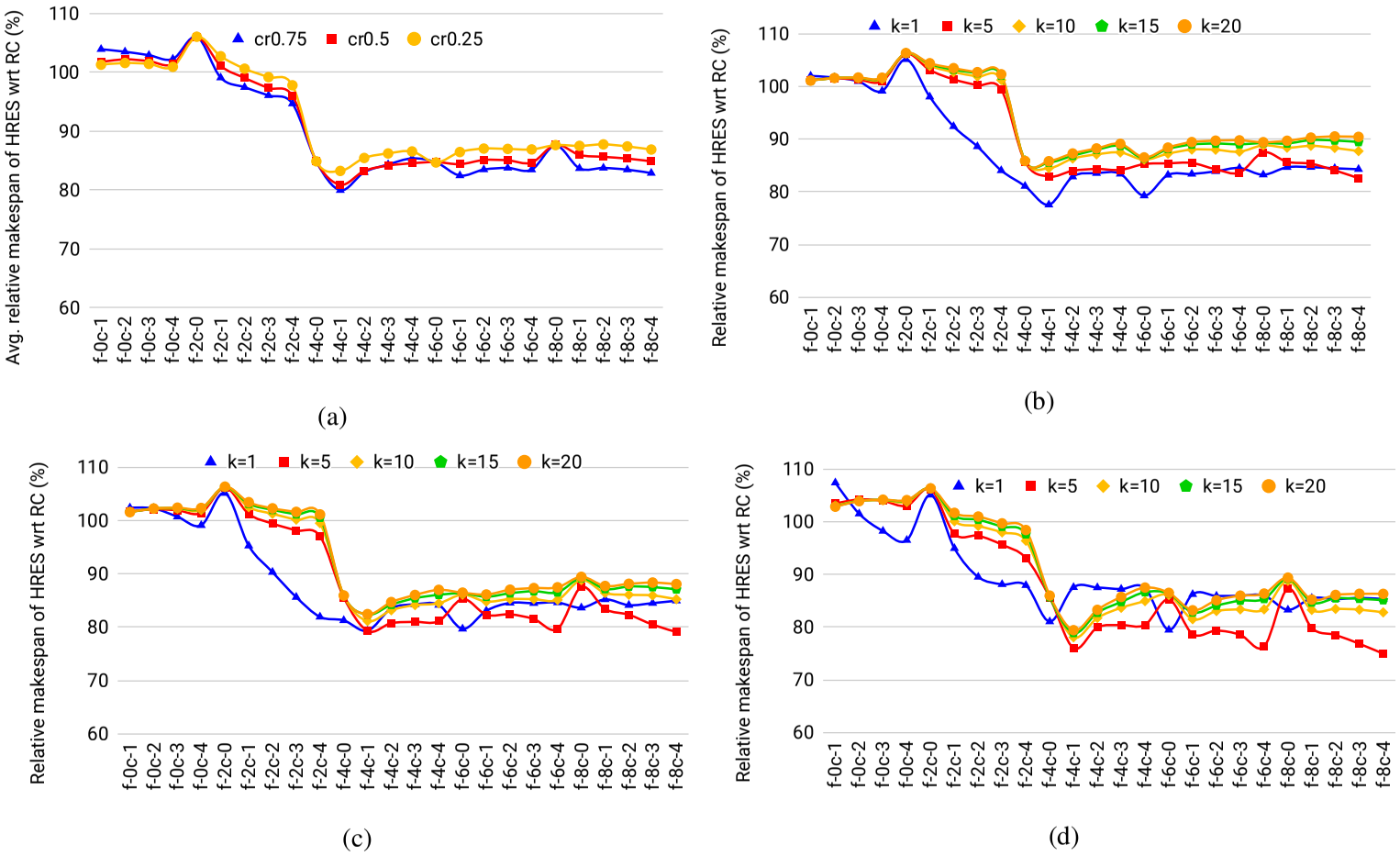


Figure 5.10: Avg. makespan w.r.t. run-to-completion for varying CPU-to-FPGA ratios is shown in (a) while their break down for ratios of 75:25 is in (b), for 50:50 is in (c), and for 75:25 is in (d). Note that k denotes arrival rate and f-*n*c-*m* denotes *n* FPGA slots and *m* CPUs.

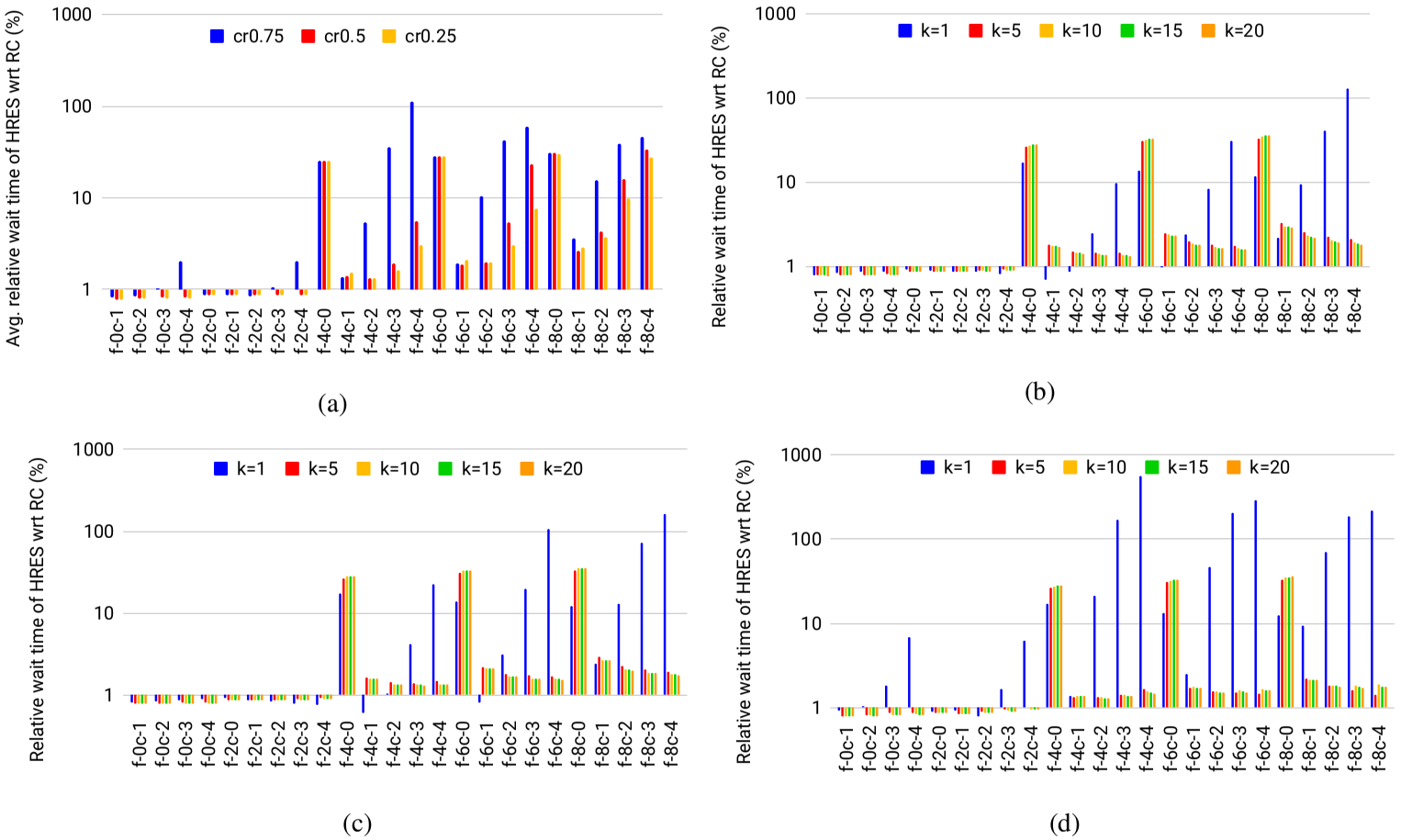


Figure 5.11: Avg. wait time w.r.t. run-to-completion for varying CPU-to-FPGA ratios is shown in (5.10a) while their break down for ratios of 75:25 is in (5.11b), for 50:50 is in ((5.11c)), for 75:25 is in ((5.11d)). Note that k denotes arrival rate and f-*n*c-*m* denotes *n* FPGA slots and *m* CPUs.

If we look in detail at the behaviour of schedulers with an arrival rate of 5 kernels and a CPU-to-FPGA ratio of 50:50, Figure 5.12 shows relative performance differences compared to standard run-to-completion model. We can see that performing context-switching without resource elasticity via methods like the round-robin can harm performance rather than improve it because the partial reconfiguration latency induces a significant overhead. Whereas, when coupled with resource elasticity, it maintains an average 20% performance advantage over run-to-completion for FPGAs ≥ 4 slots by maximising the system utilisation. However, the context-switching improves waiting time by 95% for cooperative schedulers except for HRES on pure FPGA configurations. This is due to two main reasons in this workload scenario: 1) the FPGA being overloaded leads to higher PR penalties for maximising FPGA utilisation; 2) fragmentation of resources may not allow to load a new kernel from the waiting queue onto the FPGA. In this case, our HRES algorithm takes a greedy decision and continues accelerating the current kernels by reallocating them to free resources. The need for accelerating CPU favoured kernels on the FPGA only platform further worsened waiting times. Overall, the wait time decreases if we increase the number of FPGA slots available in the system, i.e. moving from 4-slot to 6-slot and 8-slot results in a 36% and 47% lower wait time, respectively. This suggests that the ability to predict the arrival of a new kernel may be beneficial for the scheduler to avoid taking a greedy decision. Overall, for heterogeneous systems, HRES adapts better by allocating the incoming tasks to CPUs if FPGA resources are not available as a fallback mechanism, hence, reducing the wait time considerably. This results in a heterogeneous system with a better makespan than run-to-completion but also with similar wait times as round-robin policies. Note that HRES achieves this by using a better orchestration of resource allocation and not by improving any of the accelerator implementations. Overall, using both CPU and FPGA resources allows the system to deliver both fast response time (i.e. short wait time) as well as the high throughput (i.e. small makespan) of hardware acceleration.

Further, we found that all schedulers tend to keep CPU utilisation full as there are no constraints on its allocation. However, for the FPGA resources, runtime constraints can lead to fragmentation which is not dealt well by standard round-robin policies, as shown in Figure 5.13. HRES, in contrast, can use the free resources and keep the utilisation close to the maximum possible because of its ability to replicate accelerators dynamically. To perform this, HRES employs aggressive reconfiguration leading to

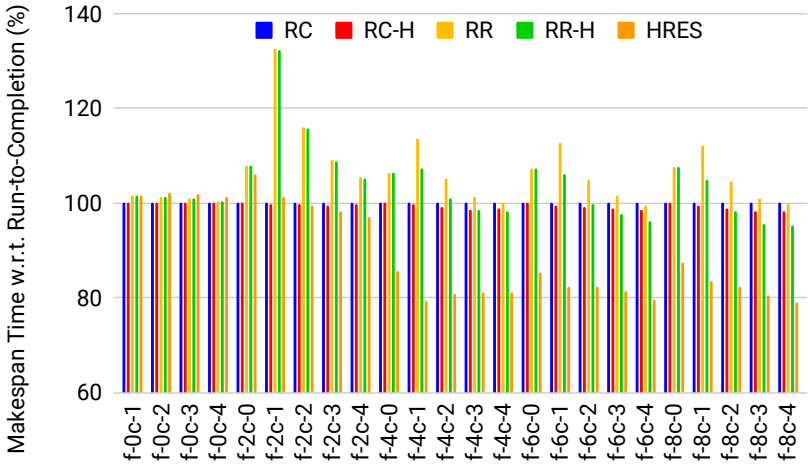


Figure 5.12: Avg. makespan w.r.t. run-to-completion for an arrival rate of 5 and CPU-to-FPGA ratio of 50:50, where f-*nc*-*m* denotes *n* FPGA slots and *m* CPUs.

a considerable increase in the number of reconfiguration calls compared to run-to-completion, as shown in Figure 5.14. However, the reconfiguration call count is similar to other context-switching algorithms such as RR and RR-H, which suggests it is more of a side-effect of context-switching itself rather than HRES in particular. Moreover, HRES performs reconfiguration only if it helps in reducing the overall makespan or accommodating a new arriving kernel. It is essential to point out that the acceleration of kernels using free resources that would otherwise be left unused amortises the higher reconfiguration penalties, yielding in a better makespan and lower wait times, as shown in Figure 5.12 and Figure 5.15.

To further quantify and contrast the overhead caused by the schedulers, we measured the average execution time of scheduler wakeup calls for each scheduler and the results of which are shown in Figure 5.16. As we can see, the overhead of HRES is $10\times$ to $100\times$ higher than that of other schedulers, particularly after the introduction of CPUs in the system. This is due to the Branch-and-Bound exploration not being able to terminate early as with FPGAs due to a lack of constraints and goes onto exploring more possible permutations for the CPU allocation. However, even in a CPU+FPGA system, this is comparatively negligible as it still requires only microseconds to compute which is marginal when compared to the PR latency, which ranges in milliseconds. It can potentially be further reduced if required by adding constraints to CPU allocation for a Branch-and-Bound mechanism based on the CPU cache preferences as a heuristic, such that we minimise redundant exploration.

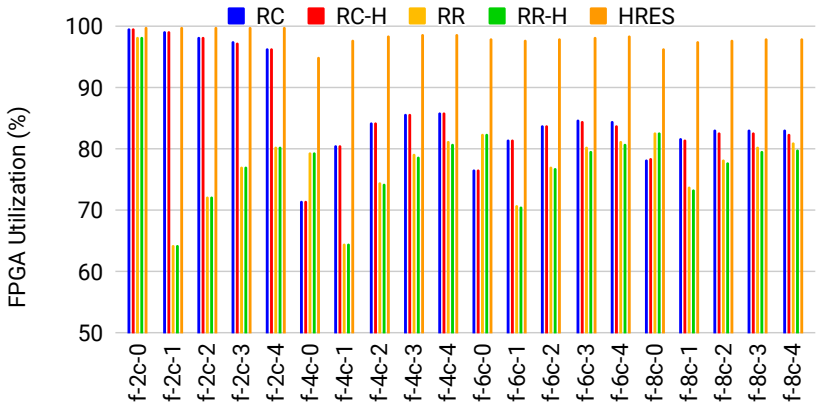


Figure 5.13: FPGA utilisation for an arrival rate of 5 and CPU-to-FPGA ratio of 50:50, where f-*n*c-*m* denotes *n* FPGA slots and *m* CPUs.

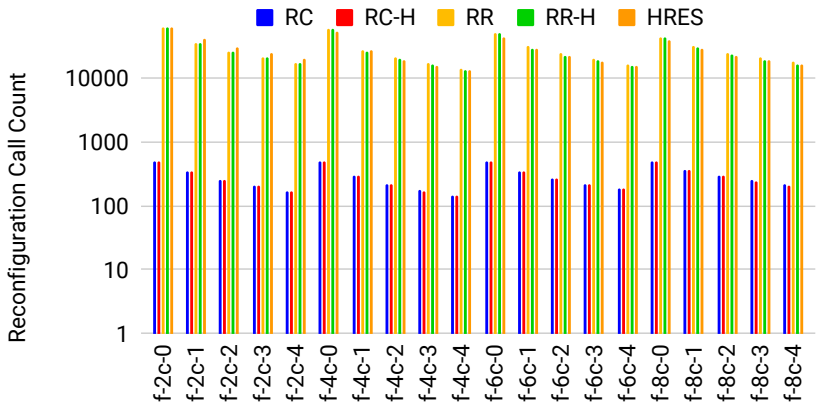


Figure 5.14: Reconfig. calls for an arrival rate of 5 and CPU-to-FPGA ratio of 50:50, where f-*n*c-*m* denotes *n* FPGA slots and *m* CPUs.

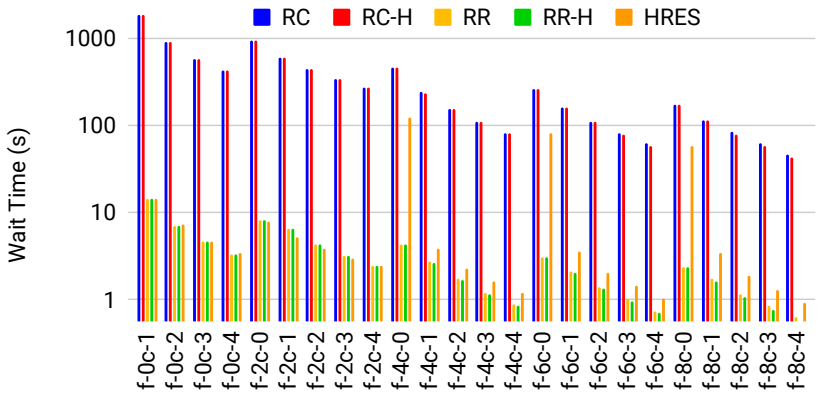


Figure 5.15: Avg. wait time for an arrival rate of 5 and CPU-to-FPGA ratio of 50:50, where f-*n*c-*m* denotes *n* FPGA slots and *m* CPUs.

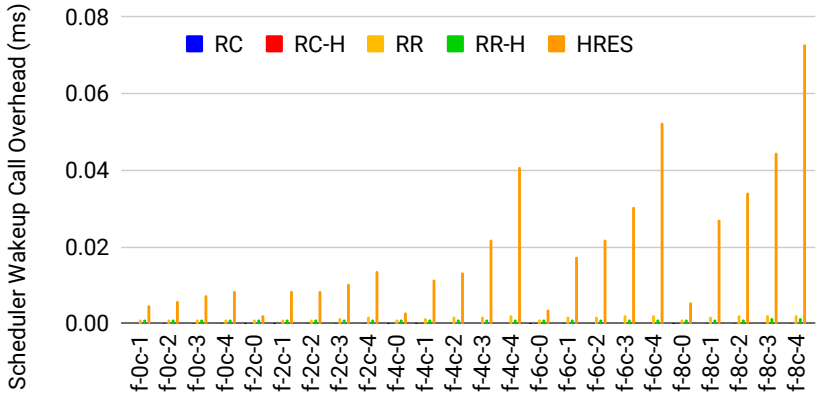


Figure 5.16: Avg. scheduler wake up call latency (time to allocation decision) for schedules with an arrival rate of 5 and CPU-to-FPGA ratio of 50:50, where f-*n*c-*m* denotes *n* FPGA slots and *m* CPUs.

5.3.3 Summary

The evaluation of our heterogeneous runtime system on a physical implementation (Section 4.3) demonstrates that a resource elastic scheduler for CPU+FPGA architectures can achieve 2× better makespan latency and 126× improvement in max waiting time compared to SDSoC-like platforms (static acceleration systems) for a set of Spector benchmarks. Moreover, our synthetic simulation experiments further extrapolate that it can provide scalable benefits of 20% performance improvement and 95% wait time improvement over the standard run-to-completion model, on various platforms across varying workload size and types.

Overall, our findings suggest five key takeaways for scheduling on future FPGA systems:

- Collaborative execution is ideal for heterogeneous resources (CPU and FPGA) and should be performed whenever possible for high performance, as it delivers fast response time and high throughput.
- Co-scheduling application on CPU+FPGA platforms can improve system utilisation and, consequently, improve performance.
- Adding a cooperative context-switching mechanism can reduce waiting time by 95%, but when coupled with fixed resource allocation policies (e.g., round-robin), it can impose significant overheads.
- Resource elasticity in almost all cases overcome the overheads of context-switching

in dynamic systems while retaining its waiting time benefits and it does not require changes in the accelerator implementation or the standard user view.

- For dynamic high workload environments, which are common in cloud and edge systems, resource elastic systems will often be a better fit than standard run-to-completion runtime systems due to its scalability.

5.4 FPGA to FPGA Accelerator Migration

After the evaluation of modular FPGA platform (Section 5.2) and resource elastic scheduling (Section 5.3) on a single node, this section will consider the scenarios where workload is migrated from on FPGA node to another using the multi-node implementation described in Section 4.4.

To evaluate the different mechanisms and scenarios for live migration, we use OpenCL kernels mentioned in Section 5.1.1 for their various compute to data ratios, as we assume the migration overhead to be highly sensitive to it. All the experiments considered in this section perform migration halfway through the application execution for a fair comparison. The effective data transfer speed between the local and remote node (both ZCU102 boards) is about 238 Mbps over the Gigabit Ethernet connection when performing TCP and therefore contributing to the main bottleneck during the data transfer phase. Our primary evaluation metric, in this section, is execution overhead caused by the migration of the accelerator. To further identify the bottleneck we split this overhead into various parts such as bitstream and data transfer, software layer overhead, partial reconfiguration and any lost work (if any) for various applications.

5.4.1 Maintenance & Load-Balancing Scenario

Migration for maintenance is often coupled with load-balancing algorithms to redistribute workload to the other nodes while the system is upgraded/recovered. Hence, in this section, we evaluate both scenarios (maintenance and load-balancing) together and highlight the critical aspects of migration concerning performance and overhead. In particular, we test the scenarios shown in Figure 5.17, where a kernel gets migrated to the target node with equal or more resources. The migrated kernel may replicate or change module implementation on the target node if possible based on our resource elastic scheduler (Section 4.3).

Figure 5.18a shows the execution overhead of the migration when performing

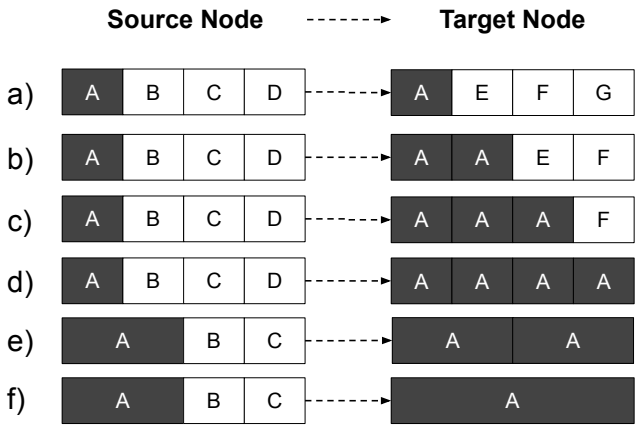
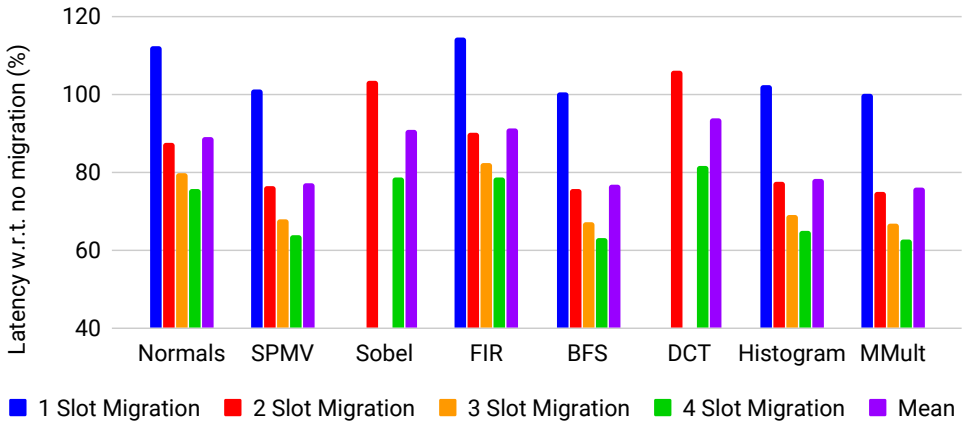


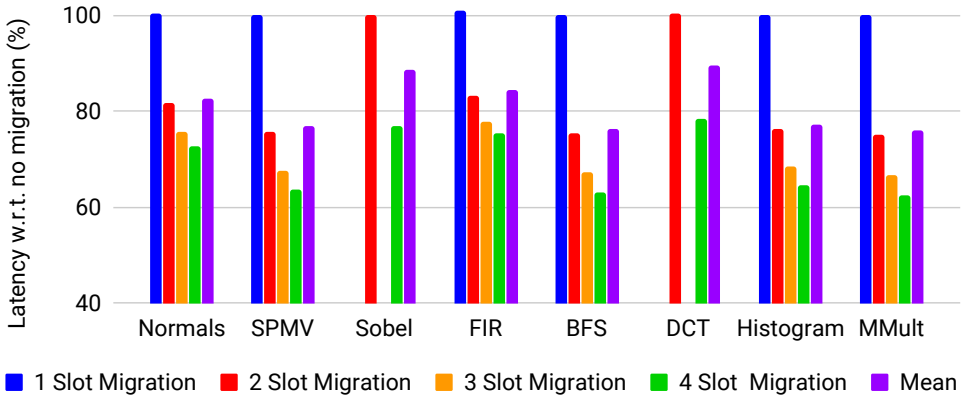
Figure 5.17: Various migration scenarios of hardware tasks on a 4-slot FPGA are highlighted: a) shows migration to an equivalently busy node, b) to e) shows migration to the target node with free resources using replication and f) shows migration to the target node with free resources with implementation alternative.

blocking migration using *replication* on the target node. Overall, it results in overheads ranging from 0.064% for Breadth First Search to 14% for an FIR filter accelerator depending upon the compute-to-data ratio. Matrix Multiplication and Breadth First Search implementations have the highest compute-to-data ratio, leading to minimal migration overhead, whereas, the opposite is true for applications like FIR and 3D Normal Estimation. In particular, when performing a migration to another node with a lower workload (allowing more resources for acceleration), we note that the overhead is paid off by the acceleration achieved for all eight applications. This implies that migration coupled with load-balancing is much more preferable than migration to an equally busy node (even when performing maintenance). However, there still exist applications like 3D Normal estimation and FIR filter, which suffer from 13% overhead on average when moving to an equivalently busy node. These latency penalties are considerable for environments such as cloud, where the service availability requirements could be strict as well as the probability of faults or need for load-balancing is high due to its large scale.

In contrast to blocking migration, non-blocking migration allows for almost zero overhead in terms of service downtime (i.e. when user computation is not taking place). Figure 5.18b shows the execution latency for the applications using replication on the target node, where the maximum overhead is 0.96% for FIR filter, consisting of the software latency and partial reconfiguration mainly. In the general load-balancing case,



(a)



(b)

Figure 5.18: The relative execution latency of kernels w.r.t. no-migration with the smallest module for blocking and non-blocking migration in (a) and (b), respectively.

when migrating to lower workload node, applications show performance benefits ranging from 17% to 37.5% using replication despite the considerable latency of data and bitstream transfer latency (as shown in Figure 5.19).

To understand the gap between these two migration mechanisms, let us consider the case of DCT migration with its implementation alternatives which require higher bitstream transfer and reconfiguration latency than other applications, as shown in Figure 5.20a and Figure 5.20b. The most significant additional latency belongs to the data and bitstream transfer over Ethernet. Thus, when utilising the non-blocking mechanism where the computation continues on the source node while all the necessary data transfer for the next work-group execution takes place, it mainly absorbs this significant latency to provide very low overhead by increasing the time spent on the source

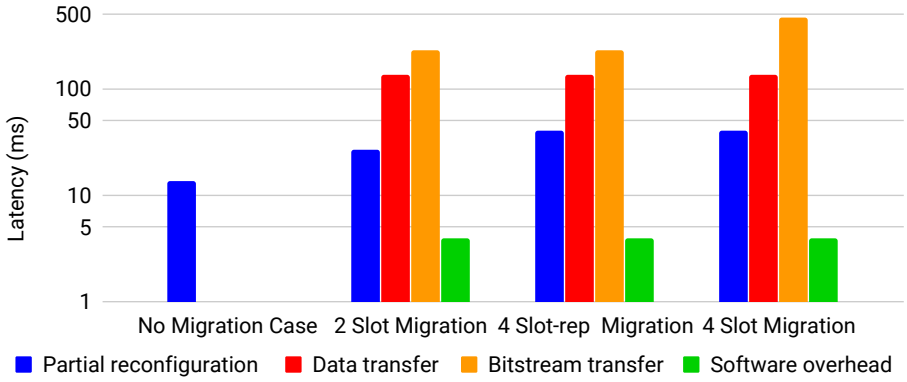


Figure 5.19: Breakdown of overhead latency involved in migration for DCT.

node. To reduce the overhead further, load-balancing can use a bigger module with super-linear performance[5] (i.e. a module taking n times the resources delivers more than n times the performance) on the target node at the cost of longer bitstream transfer latency. However, still, the work left on the target node also needs to be enough for acceleration. Example of this is the DCT (as shown in Figure 5.20b) where switching to a large module is preferable as the data transfer phase does not considerably increase the time spent on the source node.

Note that the bitstream transfer latency can be removed by keeping local copies if the workload is known. It is also possible to configure while receiving the bitstream, however, this would require a more complex protocol to handle the failed transfers.

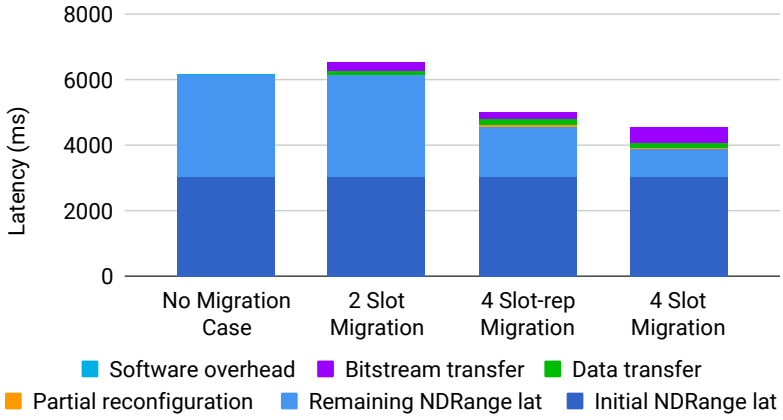
5.4.2 Fault Tolerance Scenario

Migration can also help to provide support for fault tolerance at the node level, i.e. when a node is unable to function correctly while other nodes can continue their execution/service. This fault scenario may occur due to a software bug, hardware fault (e.g., power failure) or problem in the local operating system. Hence to evaluate this, we consider two scenarios: 1) unexpected fault leading to loss of work and 2) complete system update (Linux kernel and shell together) to mitigate bugs or add new features.

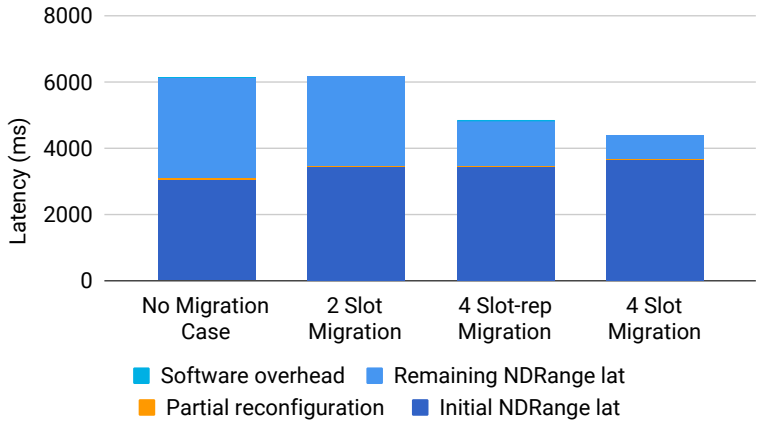
Unexpected Fault

To test the first scenario, we implemented a regular check-pointing mechanism using a server, as performed in traditional distributed systems with the help of logs [106].

[5] Better sharing of data and control logic compared to replication for many applications can gain this super-linear benefit.



(a)



(b)

Figure 5.20: Latency breakdown for blocking and non-blocking migration of DCT in (a) and (b), respectively.

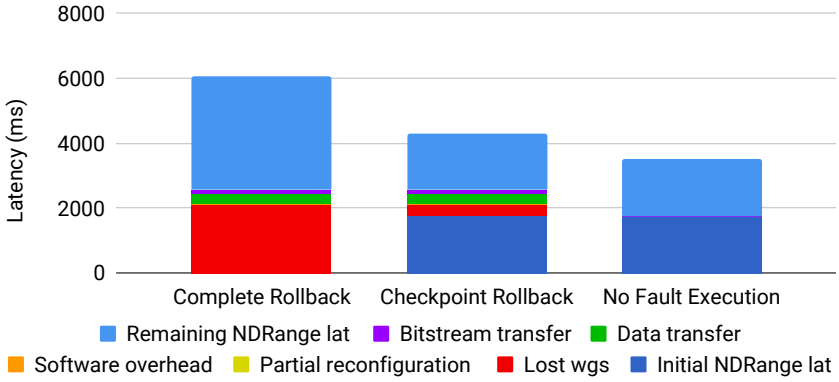


Figure 5.21: Fault tolerance scenario where an unforeseen fault occurs halfway through the execution of the 3D normal estimation application.

In our test environment, we take the snapshot of the application on every 25% of the execution length[6] based on the number of work-groups, and we mimic the unexpected fault by a forced operating system shutdown. After the fault occurrence, we resume the kernel execution again on another node.

The standard implementations of OpenCL run-times for FPGAs [24, 119] do not perform context switching and hence, they fail to leverage check-point mechanisms in case of fault occurrences, leading to a complete rollback (i.e. restart) of the kernel execution. Whereas our technique allows integrating OpenCL with the standard distributed computing mechanism such that partial rollbacks can be performed for unexpected faults (e.g., power failure). This can save up to 75% of completed work over standard OpenCL mechanisms in the here assumed scenario (as shown in Figure 5.21). This delivers an advantage proportional to the frequency of check-pointing until check-pointing overhead dominates in case of a fault occurrence. Equation 5.1 states this relation formally, where T is the total time taken for kernel execution with μ as a mean time to fault, n is the number of work-groups with a latency of L_w, D states the downtime, R is the time taken for recovery, C is the number of work-groups between check-points (i.e. check-point interval), and L_o is the check-point latency.

$$T = nL_w\left(1+\frac{1}{\mu}\left(D+R+\frac{cL_w}{2}\right)\right)+\frac{nL_o}{c} \tag{5.1}$$

System Update

In order to update the Linux kernel and static system transparently at runtime, we need to perform a series of steps. First, we need to write a new Linux image with an updated static bitstream to the SD-Card while we perform the migration. Then we need to initiate the reboot sequence on the source node using the SD-Card, followed by the initialisation of the runtime system and migration back to the source node.

Table 5.8 shows the latency of each of these steps for the Matrix-Multiplication benchmark using 4-slots via replication. The total latency for the update is 17 seconds, the majority of which corresponds to the Linux boot up period. We chose Matrix-Multiplication for this scenario as it is the longest-running application from our benchmark suite and, hence, expected the overhead induced by the update to be minimal for it. However, the update still represents 37% of the total execution time of the Matrix-Multiplication benchmark as overhead. During this period, live migration can allow

[6]This can be changed as per system requirements. The 25% of execution length is chosen here as a reasonable trade-off between frequency check-pointing and benefit provided by it for our benchmarks.

Table 5.8: Full static system and kernel update latency breakdown.

Phase	Latency (ms)
Write-to-SD-Card + Migration to Temp. Node	852
Shutdown Period	600
Boot-up Period	15160
Static Logic Reconfiguration	98
Run-time Initialisation	14
Migration to Source	278
Total Time Taken	17002

continuous provisioning of the acceleration service by moving execution to another node without pausing the kernel execution.

5.4.3 Summary

The evaluation of migration methods with the Spector benchmark suite shows that the asynchronous approach for data-parallel applications on FPGAs can allow almost zero overhead for migration and can be performed transparently from the user. In particular, this shows that we can eliminate i) the substantial latency caused by configuration read-back used by earlier proposals (e.g., 1 sec in [58]) for accelerator migration, and ii) 20% overhead caused during coarse-grained blocking migration of OpenCL applications in VOCL [121] on GPU[7].

Moreover, the ability to perform context-switch for check-pointing and migration mechanism can provide fault tolerance for FPGAs when coupled with a standard distributed system architecture. This is possible by resuming an accelerator execution from the last known consistent state in case of a fault. Consequently, this helps provide fault tolerance with fewer spare nodes (i.e. lower cost for redundancy), as the resource allocation across the cluster can be changed dynamically with low overhead. Further, it can serve as a base of dynamic load-balancing systems to provide a service with less compute nodes for saving cost and power. Lastly, it enables to perform system updates such as changing static FPGA configuration or upgrading the OS without service downtime.

[7]When compared for a single matrix multiplication application on GPU with a similar problem size

5.5 Chapter Summary

In this chapter, we evaluated the modular FPGA platform (FOS), resource elastic scheduling for heterogeneous systems and live migration for FPGA clusters. The results show that 1) modular flow can remove the unnecessary recompilation and redevelopment effort considerably while supporting multi-tenancy, 2) resource elasticity with co-operative context-switching (via data parallelism) can help improve the utilisation and system throughput while reducing wait time, and 3) live migration can be achieved at almost zero overhead and improve adaptability for FPGA clusters.

Chapter 6

Conclusion and Future Work

6.1 Contribution Summary

In this thesis, we proposed the concepts, methods, and implementations to provide better maintainability, adaptability, and accessibility for FPGA systems while using existing tools and application accelerators. With this, off-the-shelf applications in scenarios with dynamic workload had been demonstrated to 1) compile 2.34× faster and reduce update latencies by 100× over Xilinx vendor tool flow due to modular development flow; 2) run 2.1× faster with 126× improvement in max waiting time over vendor systems and on average 20% faster with 95% wait time improvement compared to other dynamic scheduling policies by introducing the concept of resource elastic execution on FPGAs; 3) migrate to other FPGA nodes with almost zero overhead via live migration.

To achieve this, we first introduced a complete modular hardware-software co-design framework, called FOS, for FPGA development and runtime management that encapsulates the heterogeneity at each level of the development process. This framework comprises all levels of FPGA development from user-facing libraries in C++ and Python for hardware acceleration in a multi-tenant system down to the FPGA shell including all the drivers.

The modular foundation allows FOS to be tolerant of heterogeneity and changes in the system components, allowing it to swap components arbitrarily as long as they maintain the component interfaces. In particular, the decoupling of the components allows the developer to adopt an application-centric view while focusing only on the part of the system they care about rather than dealing with the entire complexity involved in building heterogeneous acceleration platform with Linux. This considerably

improves the productivity for large-scale projects with multiple teams in industry [88] and academia [74], as the here introduced FOS framework allows using different tool versions, add or remove features, and do bug fixes without needing to synchronise the components with other teams. With this, FOS is bringing the FPGA development experience one step closer to software development practices.

Further, FOS provides support for traditional as well as multi-tenancy environments with low overhead and easy-to-use software interfaces. The dynamic resource allocation capabilities of FOS, allows FOS to share multiple FPGA accelerators transparently between multiple users in the time and the spatial domain as well as the ability to switch between accelerator implementations on the fly. This helps improve performance in both single-tenant and multi-tenant environments for FPGA systems (as shown in Section 5.2).

We also extended the resource allocation to include the CPU on board for OpenCL applications and solve the resource elastic scheduling problem in best-effort manner using a Branch-and-Bound algorithm with heuristics. This heterogeneous OpenCL runtime system provides the same user view as the traditional OpenCL runtime systems but undertakes dynamic allocation for *both* CPU and FPGA resources transparently from the user. With this abstraction, our runtime system then solves the optimisation problems (see Section 3.5) for the heterogeneous system concurrently based on the co-operative context-switch mechanism (via data-parallelism). In particular, it can change the accelerator implementation, device type, the number of compute units, and workload partitioning at runtime. This relieves the programmer from these optimisations at the individual application level and allows to perform the optimisations at the system level across multiple different applications automatically. Therefore, opening a new avenue for further research on scheduling for CPU+FPGA systems with collaborative execution and multi-tenancy.

It was further shown that multi-node systems built on top of modular FPGA platforms and resource elastic schedulers can benefit from better flexibility at cluster level via the live migration of accelerators. This was demonstrated through building a multi-node platform for OpenCL applications and introducing two types of live migration methods: blocking and non-blocking. The evaluation of these migration methods on various use cases (maintenance, load-balancing, and fault tolerance) showed that it can provide considerable improvement over past proposals and can be achieved with zero overhead when combined with load balancing. Hence, enabling the crucial optimisation mechanisms at the cluster level and improving integration to the existing cluster

management systems by providing similar operations known from distributed software systems.

Overall, the final system allows building scalable FPGA systems which can adapt to changing workloads (e.g. cloud and edge environments) without compromising on the programmability, performance and ease of use for FPGAs (as shown in Figure 6.1). Moreover, the benefits extend to the general FPGA users as they can equally take advantage of modular development flow and high-level user interfaces to improve their productivity.

6.2 Future Work

There are four areas of research and development which can benefit from further work: 1) improvements in accelerator compilation flow, 2) scheduler optimisations, 3) extension of FOS to PCIe based FPGA platforms, and 4) integration of FOS to software virtual machines. The details of each are as follows:

- Improvements in acceleration compilation flow include:
  - Design space exploration can help to automatically generate the implementation alternatives for the partial regions available on the target platform when using HLS. This will allow minimising the internal fragmentation of the regions and reduces the burden on developers for accelerator optimisation and further allow to use resource elasticity at its full potential.
  - Extending the HLS compilation flow to include memory DMA based context-switching logic at points in the execution where the internal state space is small based on live variable analysis for the application [11]. This would improve the adaptability of the system further by allowing preemptive context-switch and extend the resource elasticity to non-data parallel applications.
- Optimisations for the resource elastic scheduler include:
  - Extending the runtime to perform online profiling (of execution latencies) at runtime to support Branch-and-Bound exploration for generic workloads. This would allow the generic scheduler of FOS to solve the resource elastic scheduling problem more optimally.

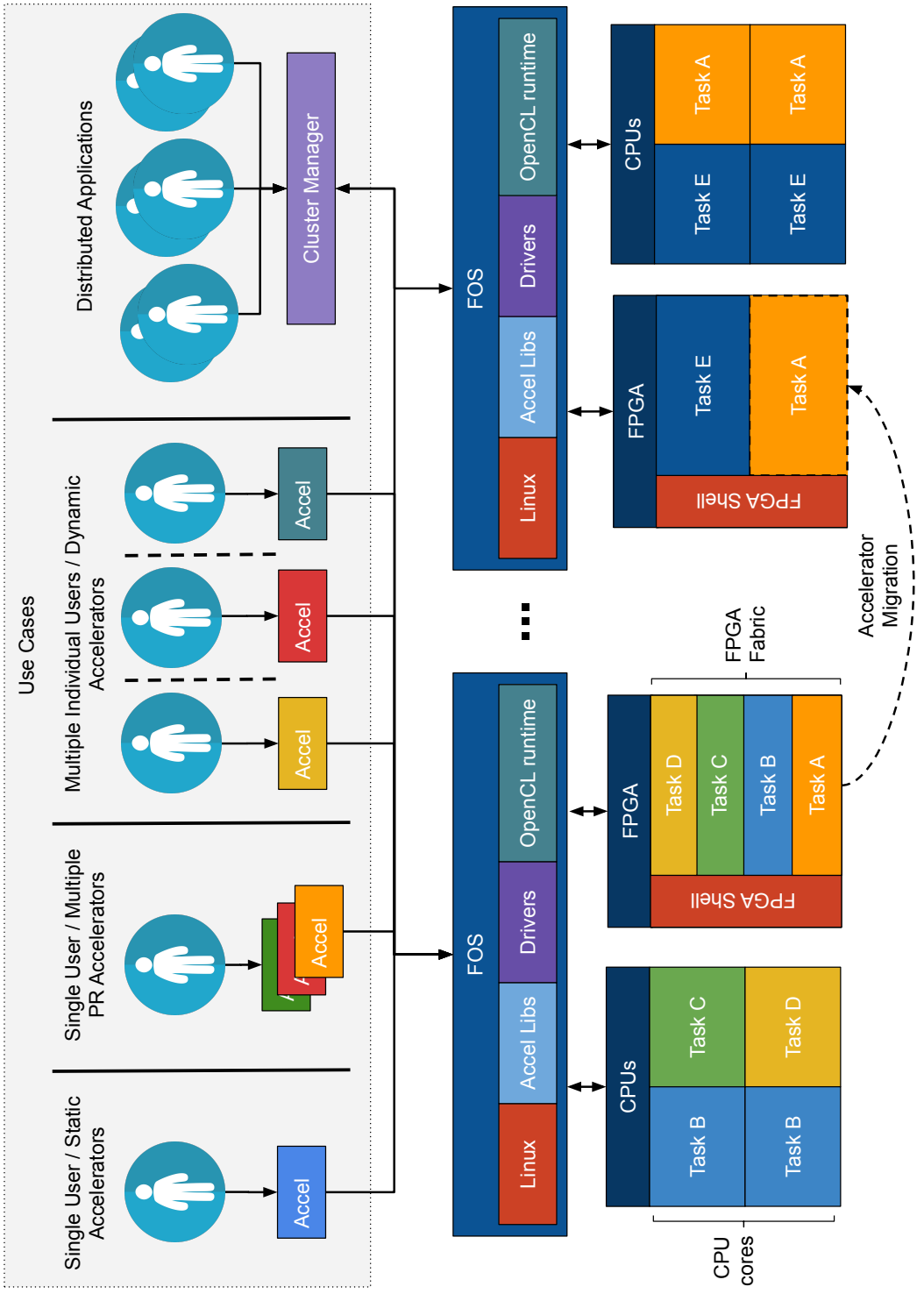


Figure 6.1: Final system (FOS shell and runtime system) with its use cases.

  - Performing scheduling with quality of service constraints for cloud optimisation goals. This would allow meeting the business and legal requirements for cloud providers while optimising resource allocation for lower operation costs.
  - Predicting incoming tasks and their resource requirements in order to preallocate resources when possible. This would allow reducing wait times further by avoiding the over-allocation pitfall encountered when accelerating only the current applications.
  - Adding memory contention and interference constraints in scheduling policies. This would allow reducing the application execution latencies caused by memory stalls or throughput deterioration due to factors such as row and bank pollution, unsuitable burst lengths and AXI port priorities [71].

- Extending the FOS framework with a soft-core and PCIe integration in the shell to support Virtex UltraScale+ platforms with one level of indirection like the Xilinx Run-time System [126]. This would allow offloading of work from an x86 host CPU and support the direct deployment of FOS in FPGA cloud services.

- Integrating the FOS into software virtual machines for better process isolation and cloud deployment support. In particular, this would complete the virtualisation support for both the software and the hardware infrastructure.

Appendix A

Theory of Resource Elastic Scheduling

To optimally solve any scheduling problem, we must first establish a theoretical description of the problem, which also provides a better understanding of its relationship with other scheduling problems and with related real-world problems. The most common approach for describing a spatial scheduling problem in computer science is the strip packing problem. Whereby the tasks are imagined as strips with the x dimension being the compute resources it requires and the y dimension being the execution latency of the task. The problem then is to pack the strips in a fixed sized x dimension (i.e. the compute resources available in the system), such that the y dimension (time to completion) is minimised. Figure A.1 shows a visualisation of one such packing example. The strips can be rearranged as well as cut to achieve a denser packing, and this models the ability of tasks to be preempted or spread out on more or fewer resources using replication. However, for resource elastic scheduling, this problem description does not model all the problem characteristics. The reason for this is that the tasks can employ implementation alternatives with super-linear performance w.r.t. the resources, implying that *the total area occupied by the strip can shrink in itself* (depending on the physical accelerator used), making the problem more complex. Hence, we need a more general problem description to understand the roots of the scheduling problem. The following Section A.1 describes the closest related scheduling problem to resource elastic scheduling and Section A.2 details the changes required to it along with a mathematical problem description that we aim to solve in this thesis.

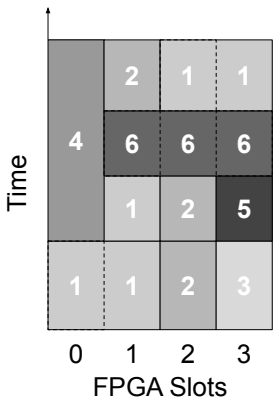


Figure A.1: Example of strip packing solution where strips can be rotated or cut using Guillotine cuts.

A.1 Relationship to Project Scheduling Problem

The Business and Operation discipline under the project planning field studies a related problem called resource-constrained project scheduling problem (RCPSP) [38, 41]. The problem aims to minimise the total makespan (also called time to completion) and schedules a set of project activities using a fixed amount of resources such that precedence and resource constraints of the project activities are satisfied. This is equivalent to scheduling computing tasks with a run-to-completion model on a computing platform. For practical scenarios, RCPSP alone is a limited model. Hence, many variants and extensions of the problem have been proposed to model practical applications of the problem more accurately. For instance, the multi-mode resource-constrained project scheduling problem (MRCPSP) [101], allows each activity to be performed in one out of several modes, where each mode has its own resource requirements and duration. An example of this is choosing to perform an activity between using automation or manual labour. In the computing world, this would be equivalent to the task being able to execute on multiple device types or number of computing units. Moreover, in real-world projects, many complex circumstances can arise, which may require interruptions. To model this, "preemption" was introduced to resource-constrained project scheduling by considering the activities to be composed of parts (sub-activities). In particular, a preemptive MRCPSP problem (P-MRCPSP) allows an activity to be preempted temporarily at any time and resumed again potentially in a different mode with *no additional cost, unlike computing platforms*. Thus, allowing the resource consumption of an activity to be changed dynamically in time as required by changing the mode after each part of the activity.

A.2 Resource Elastic Scheduling Problem

Resource elastic scheduling (RES) can be modelled as a project scheduling problem variant with multi-mode and preemption extensions. The modes in which an activity can be performed can be considered equivalent to the implementation alternatives a task has. Similar to different activity modes, each implementation alternative can have different time, area, energy and throughput characteristics. Preemption of an activity is similar to an accelerator context-switch for accelerators with the *additional constraint of given minimum execution time* before the preemption can be performed (i.e. to reach a quasi-quiescence point with a small state space in an execution flow). After a context-switch, a task can potentially be reloaded onto an FPGA with a different implementation alternative in a similar manner as an activity can change its mode after preemption, except that changing to another implementation alternative imposes a *context-switch overhead* (time penalty) depending on the type of change (e.g. changing the allocation from one instance of an accelerator to two or more). These differences require modifications in standard P-MRCPSP constraint definitions (as revealed in the following Section A.3).

A.3 Notation and Problem Formulation

A resource elastic scheduling problem analogue to the P-MRCPSP problem can be described as a graph $G(V,E)$, where V is the set of nodes representing tasks, and E is set of arcs representing execution dependencies. Dummy tasks 0 and $n+1$ act as the beginning and end of the schedule. Each task in $i \in V$ is made up of parts (sub-tasks) p which can be executed in mode $m \in M_i$, where a mode m has its own time (d_{im}) and resource requirements ($r_{imk,p}$). Dummy tasks $i = 0$ and $i = n+1$ have zero duration and resource requirements. For resource k, the availability of R_k is constant and is constrained throughout the scheduling problem. Cooperative preemption of tasks incurs a time penalty (O_i) depending on the overhead associated with the change from mode M^p to M^n for the tasks. The maximum number of context-switches allowed is U_i, which is the maximum number of parts of a task. For each task i, a minimum execution time ε_{im} is defined during which a task i in mode m cannot be interrupted. Mode M^0 and M^{U_i+1} represent the uninitialised state of the resources (i.e. no resource allocation). Table A.1 summaries the notation and parameter definitions used for this formulation.

Table A.1: Notation and parameter definitions for resource elastic scheduling problem description.

Sets		
V	A set of task supporting cooperative scheduling	
E	A set of arcs representing execution dependencies	
M_i	A set of modes for tasks i (i.e. implementation alternatives)	
Indices		
i, j	Index of tasks	
m	Index of execution modes	
k	Index of resource type	
p	Index of task parts	
Parameters		
n	Number of tasks to schedule	
$0, n+1$	Dummy starting and ending tasks	
U_i	Maximum number of task parts (sub-tasks) for a task i	$i \in V$
$U_i + 1$	Dummy part of task i marking the end of execution	$i \in V$
α_i	Maximum interruption time for task i	$i \in V$
$\|M_i\|$	Number of execution modes for task i	$i \in V$
$r_{imk,p}$	Resource required of type k for part p of task i in mode m	$i \in V;\ m \in M_i;\ p = 1,2,...,U_i$
R_k	Maximum resource units available of type k	
$\theta_i(M^p, M^n)$	Switching latency of task i from mode M^p to mode M^n	$i \in V;\ M^p, M^n \in M_i$
d_{im}	Duration for task i in mode m	$i \in V;\ m \in M_i$
Decision variables		
$S_{i,p}$	Positive decision variable: Start time for part p of task i	$i \in V;\ p = 1,2,...,U_i$
$F_{i,p}$	Positive decision variable: Finish time for part p of task i	$i \in V;\ p = 1,2,...,U_i$
$x_{im,p}$	Binary decision variable: is 1 if part p of task i is executed in mode m; and 0 otherwise.	$i \in V;\ m \in M_i;\ p = 1,2,...,U_i + 1$
$t_{im,p}$	Positive continuous decision variable: Duration for part p of task i in mode m	$i \in V;\ m \in M_i;\ p = 1,2,...,U_i$

The following mixed integer non-linear programming formulation models the resource elastic scheduling as a variant of the P-MRCPSP problem with context-switching overhead:

$$\text{Min } Time = S_{n+1,0} \tag{A.1}$$

The relation (A.1) describes the minimisation of the makespan as an objective. such that

$$\sum_{m=1}^{|M_i|} x_{im,p} = 1; \quad \forall i \in V;\ p = 0,1,...,U_i \tag{A.2}$$

Constraint (A.2) enforces that part p of task i can only exist in one mode at a given time.

$$\varepsilon_{im} \leq t_{im,p} \leq d_{im}; \quad \forall i \in V;\ p = 0,1,...,U_i + 1 \tag{A.3}$$

Constraint (A.3) models that there is a minimum execution time ε_{im} and a maximum execution time d_{im} during which the cooperative task i in mode m is in progress without interruption.

$$S_0, F_0 = 0 \tag{A.4}$$

Constraint (A.4) establishes that there exists F_0 and S_0 as dummy parts to simplify the formulation in (A.5). Note that these dummy parts are defined to be of zero latency in Table A.1 and, hence, does not affect the actual solution space.

$$F_{i,p}, S_{i,p} \geq 0; \quad \forall i \in V;\ p = 0,1,...,U_i + 1 \tag{A.5}$$

$$x_{im,p} \in \{0,1\} \quad \forall i,m,p \tag{A.6}$$

Constraints (A.5) and (A.6) guarantee the start and finish time are positive and decision variable is binary, respectively.

$$F_{i,U_i} \leq S_{j,0}; \quad \forall (i,j) \in E \tag{A.7}$$

Constraint (A.7) ensures that the earliest start time of task j is forbidden to be

smaller than the finish time of its predecessor task i.

$$\sum_{p=1}^{U_i} \sum_{m=1}^{|M_i|} \left(\frac{t_{im,p}}{d_{im}} \right) x_{im,p} = 1; \quad \forall i \in V \tag{A.8}$$

Constraint (A.8) ensures that the sum of the relative progress of all parts of a given cooperative task in all execution modes is equal to one unit task, i.e. task executes entirely.

$$\sum_{i=p_t} \sum_{m=1}^{|M_i|} x_{im,p} r_{imk,p} \leq R_k \quad p = 1,2,...,U_i;\ k = 1,2,...k$$
$$p_t = \{ i \in V \mid S_{i,p} < t \leq S_{i,p+1} \}; \tag{A.9}$$

Constraint (A.9) enforces resource limit of R_k, for each time instant t, and for each resource type k.

$$O_i(M^p, M^n, p) = \begin{cases} 0, & \text{if } M^n = M^0 \text{ or } p = 0. \\ \theta_i(M^p, M^n), & \text{otherwise.} \end{cases} \tag{A.10}$$

Equation (A.10) defines the overhead (e.g., for FPGA reconfiguration) of changing from mode M^p to mode M^n for part p of task i.

$$F_{i,p} - S_{i,p} = \sum_{y=1}^{|M_i|} \sum_{m=1}^{|M_i|} \left(x_{im,p} t_{im,p} + O_i(yx_{iy,p}, mx_{im,p}, p) \right)$$
$$\forall i \in V;\ p = 0,1,...,U_i + 1 \tag{A.11}$$

Constraint (A.11) checks that the duration of the part p of task i in mode m should be equal to the difference between the finish ($F_{i,p}$) and start ($S_{i,p}$) time for part p of task i plus the penalty of changing the mode if any (O_i). It models the execution latency of the task plus the overhead imposed by context-switching to another implementation of the task.

A.4 Mapping Heterogeneous Systems to RES problem

In a heterogeneous environment where the resource type can defer (i.e. CPU, GPU, FPGA, ASICs), the scheduling can become challenging while optimising for performance as each task can have preferred execution mode (e.g., Task A might perform better on CPUs than FPGAs while another task can have the opposite behaviour). In particular, moving from one device type to another can have varying latency depending on the data transfer, setup time, and minimum execution length required for the target device.

We can model these scheduling requirements using the RES problem defined in Section A.3. In which heterogeneous tasks ($i \in V$) can run on different devices (mode M_i) each with different latency ($t_{im,p}$) and particular type of resource requirement ($r_{imk,p}$), where the availability of the resource type (device type) at any given time instance is bound by the heterogeneous system configuration (R_k). The cooperative nature of the problem allows changing the mode during the execution at the end of each part given that the minimum execution length (ε_{im}) constrained is satisfied. Further, the penalty of changing from one mode to another mode ($O_i()$) models the changing of execution device or mode of execution on the same device. Note, it is also possible to share the device between tasks as the resource capacity (R_k) can be more than 1, allowing task modes such as multi-threading for CPUs. Further, tasks can have a specific type of resource requirements ($r_{imk,p}$) and may not necessarily have mode m for each resource type (i.e. a task may run on a CPU and GPU but not on FPGA). With this heterogeneous extension, the objective is again minimising the total execution time, as stated by relation (A.1).

Note, since resource elastic scheduling for both FPGA alone and heterogeneous systems map to the same problem, they are theoretically equivalent problems. This implies that the complexity of scheduling for FPGA resources alone is equivalent to heterogeneous scheduling for a platform with FPGA resources. This complies with common expectations given that FPGA can be used to model any other device (e.g. CPU or GPU) as a digital circuit.

A.5 Discussion

The resource-constrained project scheduling based formulation in Section A.3 shows that the RES problem shares its root with scheduling problems from the business and

operation discipline and that it can be considered theoretically equivalent to the project scheduling for software team or large scale government projects [38, 41].

However, since the RES problem is a variant of resource-constrained project scheduling problem and more complex than strip packing problem, it is an NP-hard problem [9, 40]. Implying there only exists algorithm with exponential complexity for its solution, unless $P = NP$. The conventional approach to solving a mixed-integer non-linear programming formulation like the one presented in Section A.3, is to use a Branch and Bound technique with a combination of standard non-linear optimisation methods. Whereby the technique begins by recursively breaking down the problem into multiple ILP sub-problems until a solution is found that satisfies all integer constraints, resulting in exponential search space and execution time. Overall, finding a solution using this method can take anywhere between milliseconds to days, depending on the complexity of the problem instance.

Consequently, for a dynamic system where tasks can arrive at any time without prior notice, this exploration needs to take place at runtime, which makes it infeasible for our purposes. Hence, there is a need for domain-specific heuristics to simplify the problem and solving it as close to the optimal solution as possible. To this end, we propose solving the problem in snapshots, i.e. allocating the resources given the runtime conditions at the scheduling event and not looking into the future for an entire scheduling trace. This is because computing the schedule ahead of time may become obsolete if a new task can arrive at any time. Overall, this simplifies the problem into a variant of the bin-packing problem by removing the time dimension and not considering dependent waiting tasks. With this, we can solve the resultant bin-packing problem optimally using standard Branch and Bound technique. However, to find an approximate solution for makespan optimisation of the RES problem, we still need to link these individual snapshot solutions. To achieve this, we can use domain-specific heuristics to estimate the resulting makespan latencies at each step. In Section 4.3.3 provides an implementation of one such Branch and Bound technique with the snapshot heuristic and ranking functions (domain-specific heuristics) to solve the RES problem at runtime.

Appendix B

Resource Elasticity for Long and Short Running Tasks

This appendix presents a *predecessor* to the runtime system presented in Chapter 4 and uses a brute-force approach based on fairness and performance for OpenCL applications on FPGAs. This runtime system is then used to evaluate resource elasticity for long and short running task workloads using a behavioural simulator and a case study with in-house accelerators. Note that this legacy system [109] is only presented as additional information on how a resource elastic scheduler may behave for fairness requirements and workload of long and short running tasks.

B.1 FPGA Virtualisation with OpenCL

We selected the OpenCL execution model [77] for a case study as it is an industry-standard for High-Level Synthesis (HLS) and heterogeneous computing in general (see Section 3.5.2 for details). We describe the design details of our runtime resource manager for performing resource elasticity in the following subsection.

B.1.1 Resource Manager Design

We have implemented a resource manager that consists of four different components: Waiting Queue, Scheduler, PR Manager and Data Manager, as shown in Figure B.1. The Waiting Queue keeps track of kernels waiting for execution and contributes to the implementation of a Round Robin policy for time-domain multiplexing (TDM) if the demand for accelerators exceeds the available resources (typically given in terms of

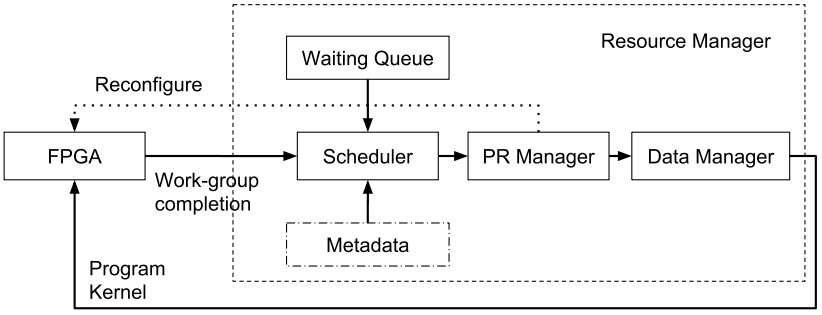


Figure B.1: Resource manager architecture for virtualizing the resource footprint of the OpenCL kernels at runtime.

slots available). The scheduler performs Space Domain Multiplexing (SDM), i.e. it decides which modules to expand or to shrink and in what manner given some meta-data (profiling information and available bitstreams) of kernels. Further, the scheduler is also responsible for identifying which module should be replaced when performing TDM. The PR Manager performs the partial reconfiguration requests issued by the scheduler. While the Data Manager keeps track of the work-group execution for each kernel; it is also responsible for programming the accelerators for the next work-group. Note that the whole flow only triggers when a host program issues an execution command or if a kernel finishes its work-group execution.

This model shares some ideas of a cooperative operating system that can guarantee real-time behaviour in the absence of a global (interrupt) timer. Because the execution of a work-group requires that all data is available and because execution time of work-items is commonly not data-dependent (best practice), upper execution time-bound can be defined, which allows the application of resource elasticity under real-time constraints.

Upon the scheduler wake up call, the resource manager retrieves data from all accelerators which have completed their work-group execution and goes through the following stages:

Kernel Selection

At first, the resource manager selects the set of kernels which need to be considered for the allocation of resources. This mainly depends on three cases based on the waiting queue size and FPGA state. i) When the FPGA is empty, and the waiting queue is non-empty, it extracts the maximum number of kernels which can be run on the FPGA concurrently from the queue, based on their minimum size modules. ii) When

the FPGA is non-empty, but the waiting queue is empty, the selected kernels are the kernels currently running on the FPGA. iii) When the waiting queue and the FPGA are non-empty, it performs Round Robin scheduling in the time domain by closing all the kernels (removing them from the FPGA) which have completed their work-group execution and inserts them back to the waiting queue. After this, it shrinks currently running kernels by employing a heuristic for recovering the maximum possible slots. We implemented this heuristic to be fair to all the waiting kernels by allocating resources to them as soon as available. It is possible to replace this heuristic with another to cater to other scheduling aims such as performance or energy. Note that at this stage of scheduling, we may have kernels which have one or more accelerators currently executing and that cannot relocate. In this case, we select those kernels plus the maximum number of kernels from the queue which can be executed together, as given by the free slots available.

Generation of Layout Options

Once the kernels are selected, the module layout is computed (i.e. the exact position and size for each accelerator) for the selected kernels. This problem has a large search space as each kernel may have different implementation alternatives available (i.e. different accelerator sizes) which may also be replicated, hence increasing the number of possible combinations. However, this has three constraints which substantially prune the search space:

1. *Availability* of only certain sized modules rather than all implementation alternatives (e.g., a kernel may only have modules which may take at least 2 slots, making it infeasible to be used when only one free slot is available).
2. *Runtime constraints*, i.e. executing kernels are not cannot relocate (due to cooperative context-switch), so this may constraint the available free space.
3. *Over allocation*, i.e. we cannot allocate more slots than the FPGA can provide.

Currently, we perform this computation by enumerating all possible combinations and removing infeasible module layouts based on these three constraints. Note, since the number of slots on an FPGA tends to be small, the enumeration space will be small as well. However, one may employ heuristics for better runtime at the risk of losing the optimal solution.

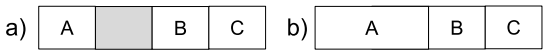


Figure B.2: Logical slot module layout example where a) is completely fair but wastes one of the slots, while b) is not absolutely fair but presents an acceptable trade-off with better utilisation.

Resource Allocation

After the layout generation, the resource allocator evaluates module layouts based on fairness with an adapted version of the Jain index [54] which accounts for the utilisation of slots. Note that with *fairness* we refer to the resource allocation for a set of kernels only. Currently, we do not consider the time domain for fairness. Equation B.1 gives the adapted version of the Jain index, where x is the set of kernels in a module layout, n is the number of kernels and x_k denotes the slot allocation for kernel k in the module layout. We added the term $u(x)$ for stating the number of slots utilised by the module layout and which is calculated by Equation B.2. We need to consider utilisation ($u(x)$) for fairness because absolute fairness may come at the cost of resource wastage. Consider the example shown in the Figure B.2: if we were to choose module layout a) we would be completely fair in allocation between kernels by giving each kernel a single slot. However, we would be wasting one of the available slots. If, instead, we choose module layout b) we would not be completely fair but would have an acceptable trade-off for maximising resource utilisation and performance.

The resource allocator selects the module layout with maximum score, which is formalised by Equation B.3. Where $r(x,n)$ is the fairness score of the module layout x and L_f is the set of all feasible module layouts.

$$r(x,n) = \frac{(\sum_{k=1}^{n} x_k)^2}{n \times \sum_{k=1}^{n} x_k^2} + u(x) \tag{B.1}$$

$$u(x) = \sum_{i=1}^{n} x_i \tag{B.2}$$

$$\arg\max a(x) = \{r(x,n) \mid x \in L_f\} \tag{B.3}$$

Resource Binding

After the allocation of resources for each kernel, the resource binder identifies the best possible module and number of instances for it, in terms of performance considering

the resource constraints based on the ranking function (Equation B.4). We project the performance possible for each implementation alternative by calculating the time to completion of each kernel based on Equation B.4, where x_k is the number of slots allocated to kernel k; $T_c(m)$ is the time to completion of module m; $P(m)$ is the time taken by partial reconfiguration for module m; and L_f^x is the possible combination of modules and number of instances for a kernel k with the resource constraint of x_k. The time to completion for a given module can be calculated by using the information provided by the HLS tool, profiling or annotated by the programmer as meta-data. For instance, an OpenCL kernel synthesis run can derive the information of the work-group execution time ($T_w(m_k)$) while a programmer can highlight work-group size (W_s). Here, the completion time is *estimated* based on Equation B.5, where $W_l(k)$ is the number of work-items left for the execution of Kernel k and which is known by Data Manager. The partial reconfiguration cost is specific to the FPGA used and generally scales linearly with the number of slots. Equation B.6 models the configuration cost, where $P(m)$ is the total latency for partial reconfiguration of module m; P_s is the configuration bitstream size of a single resource slot; N_m is the number of slots required by module m; and P_t is the throughput of the configuration port (e.g., the Internal Configuration Access Port - ICAP) of the FPGA.

$$\arg\min p(x_k) = \{T_c(m) + P(m) \mid m \in L_f^x\} \tag{B.4}$$

$$T_c(m_k) \approx T_w(m_k) \times \frac{W_l(k)}{W_s(m_k)} \tag{B.5}$$

$$P(m) = \frac{P_s \times N_m}{P_t} \tag{B.6}$$

Our projection of performance takes into account the acceleration possible with a given module (by calculating its time to completion) and overhead of partial reconfiguration to help tackle the Trade-offs 1 and 2 as mentioned in Section 3.4.1. The same technique is used to break the tie (when multiple fair solution exists) in favour of the first best performing configuration when ranking is performed by the resource allocator. After that, the resource binder requests PR Manager to load the new module layout.

Partial Reconfiguration Manager

Upon receiving a reconfiguration request, the PR manager performs partial reconfiguration for each instance of a kernel one at a time based on the module layout selected by the resource allocator and resource binder. After reconfiguration, PR Manager instructs the Data Manager to provide input to the new accelerator.

Programming Kernels

Data Manager programs the kernels with new input data and also retrieves the output data when available. Further, it handles the case when a kernel is changed to a differently sized implementation alternative which leads to a change in work-group size, such that the outcome of the kernel remains the same. It does this by calculating the next work-group indices such that no work items are left out at a new granularity. This may require certain work-items to be re-evaluated. However, this is safe to perform as kernels commonly write their output at a different memory location from where the input operands are stored and as each implementation alternative provides the exact same functionality for each work-item.

This section revealed the resource manager which provides a reference implementation for the operating system services which can consider various resource elasticity trade-offs for running OpenCL accelerators. While other heuristics may be tailored to meet specific system requirements (e.g., performance, response time, power consumption), a runtime manager for a resource elastic system always has to provide the space-time mapping of resources, perform reconfiguration, and managing the execution of kernels.

B.2 Simulation Experiments

To evaluate the characteristics of the scheduling algorithm, we conduct a series of simulation experiments to explore how different scheduling decisions impact our resource elastic virtualisation approach. Section B.3 presents a practical case study while the following subsections describe the simulation experiments and findings in detail.

B.2.1 Experiment Setup

With simulation, we explored the effects of scheduling on a wide variety of applications by changing the characteristics of synthetic applications in terms of area requirements and completion time. This was used to further investigate scheduling effects on different sized FPGAs by varying the slots available for scheduling.

We tested the runtime system with 1000 different compute-bound scheduling requirements, where randomly arriving and terminating tasks run in parallel. To model a compute-bound schedule, we generate a long-running kernel of 4000 work-groups at the arrival time zero, while we generate the other accompanying kernels in the schedule in the range of [1, 12] for each experiment. Except for the arrival time and the number of work-groups of the long-running kernel, characteristics of each kernel are generated randomly. Where the base latency of work-groups is chosen in range of [10, 100], the arrival time is in [1, 10000], the minimum slot size of kernels is in the range of [1, min(4, n-1)] where n is the total number of slots available on the FPGA, and the maximum slot size of a kernel is set [minimum slot size, min(4, n)]. The speedup achievable with different sized kernels is chosen between [3, 10] and the number of work-groups is in [50, 500]. The respective parameter selection is based on a uniform random distribution. For all experiments, we assume the I/O requirements of kernels are not on the critical path of the application. The partial reconfiguration cost is modelled linearly proportional to the number of slots with the cost of configuring a single slot being $5\times$ the smallest latency possible for a work-group. Note that the given range of work-group latency and speed up available from configuring another kernel will likely not recover the cost of reconfiguration from a single work-group execution which is the anticipated behaviour in a real-world scenario with highly optimised accelerators. Furthermore, restricting the kernel size to a maximum of 4 slots allows studying a scenario where a user runs kernels of given sizes on a much bigger FPGA. The scenario assumed here look similar to the example in Figure 3.6b showing a fine-grained schedule and consecutively relatively high configuration overhead. While the configuration cost is the number of slots to be reconfigured over time, the relative overhead depends on how long the module runs after configuration. This means that the coarser the scheduling granularity is, the lower the relative configuration overhead. Further, it is essential to note that our simulation workload shares some similarity with workload traces found in Google clusters [90], with its mix of long and short running kernels.

We ran the randomly generated schedule requirements on five different schedulers.

The first three schedulers act as a baseline for our implementations, and these are as follows:

1. **Normal Scheduler (NS)** which allocates the tasks to a First-Fit slot and execute them in run-to-completion mode. This is the most commonly used strategy [13, 16, 67] and is beneficial for FPGAs as the reconfiguration latency is high.
2. **Conservative Cooperative Scheduler (CCS)** is a time-domain scheduler which performs a context switch when the task voluntarily relinquishes the control (in our case at the end of a work-group execution). However, since standard co-operative schedulers do not take into account the availability of implementation alternatives, it may always use the minimum sized module to operate in conservative mode. This strategy aims at minimising the number of reconfigurations required as it would leave maximum space available for new incoming kernels.
3. **Aggressive Cooperative Scheduler (ACS)**, in contrast to CCS, employs the biggest module available for achieving maximum possible performance at the risk of higher reconfiguration count.

Our two implementations of resource elastic scheduling are 1) Standard Resource Elastic Scheduler (SRES) and 2) Performance-driven Resource Elastic Scheduler (PRES). The implementation of SRES is discussed in Section B.1.1. PRES follows the same implementation, but resource allocation ranking is performed to maximise performance. Equation B.7 captures the ranking used for PRES by minimising the total completion time for the kernels arriving at runtime, where $T_c(x_i)$ is the completion time calculated using Equation B.5, x is the module layout, and L_f is the set of all feasible module layouts.

$$\arg\min a(x) = \{\sum_{i=1}^{n} T_c(x_i) \mid x \in L_f\} \tag{B.7}$$

B.2.2 Results

The average completion time (the most relevant performance indicator) for each scheduler is shown in Figure B.3 and the average waiting time for each kernel is shown in Figure B.4. The wait time is calculated as $w_i = s_i - a_i$, where a_i is the arrival time of the kernel and s_i is the time when it begins its execution (after partial reconfiguration). The completion time is the lowest for NS of all the baseline schedulers for small FPGAs as it does not have to pay the higher reconfiguration penalties compared to the other

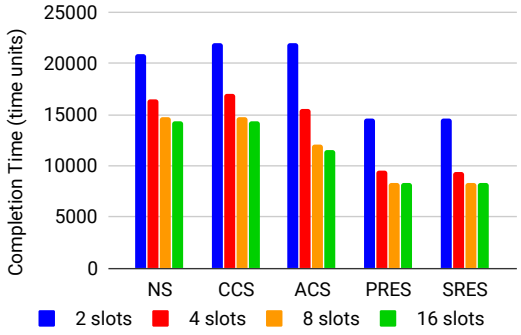


Figure B.3: Average completion time for each scheduler on FPGAs with 2, 4, 8, and 16 slots.

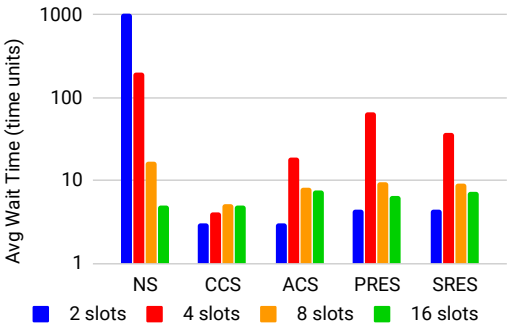


Figure B.4: Average wait time for a kernel on FPGAs with 2, 4, 8, and 16 slots.

schedulers. Note that despite the better performance, NS has poorest wait time from all the baseline schedulers, as it lets all kernels run to completion without interruption. In contrast, resource elastic schedulers provide considerably lower completion time and similar wait time characteristics as cooperative schedulers (Figure B.4) while also incorporating higher reconfiguration overhead. In particular, if we compare the performance targeting versions of schedulers, i.e. PRES and ACS, we can see that PRES can achieve a higher performance of as much as 39% to 64% over ACS. However, the performance advantage does not scale linearly with the number of resource slots available as they become so abundant that all kernels can run concurrently in their full-sized implementation alternatives for most of the experiments without higher reconfiguration cost when using ACS. Resource elastic schedulers achieve this performance advantage by considering all the possible implementation alternatives and employing them dynamically at runtime to maximise performance.

We measured the resource utilisation over the total number of slots occupied after every scheduler wake up call, as plotted in Figure B.5 after normalisation. We can see that the employment of ACS leads to the highest utilisation from baseline due to the heuristic of always using the largest module for each kernel. However, this does not tend towards the maximum possible utilisation as it cannot overcome the fragmentation repercussions of its heuristic. While on the other hand, the ability to adjust the allocation of resources and replication of modules for higher performance helps our resource elastic schedulers to gain almost full utilisation, which is about on average 2.3× higher utilisation compared to ACS and about 2.7× better on average when compared to NS. The drop in utilisation with respect to an increase in the number of slots available occurs because resource elastic schedulers tend to employ the biggest

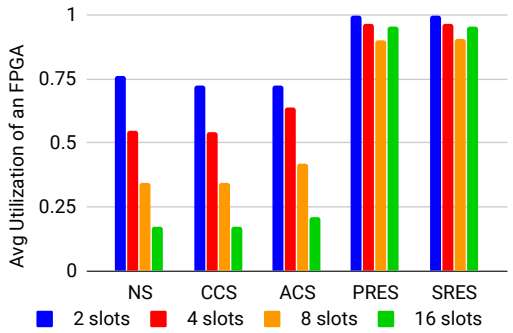


Figure B.5: Average utilisation of an FPGA when measured at the end of a scheduler's wake up call for FPGAs with 2, 4, 8, and 16 slots.

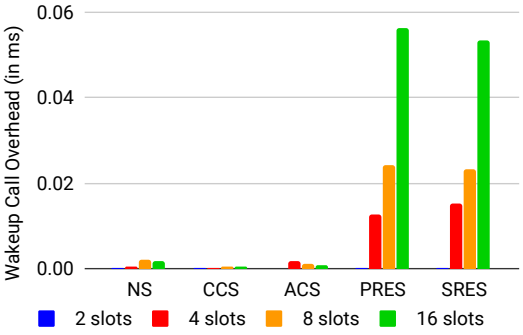


Figure B.6: Average scheduler execution time for wake up calls on FPGAs with 2, 4, 8, and 16 slots. Note, certain bars are not visible due to the their relatively small values.

module possible for a kernel to maximise performance at the cost of fragmentation and the scenarios where kernels did not offer smallest module size for filling up the holes left in a particular module layout. Note that the higher utilisation is achieved at the cost of a higher number of reconfiguration calls; the implication of this is captured in Figure B.3 and B.4, as reconfiguration time is included in the total completion and wait times.

To quantify and contrast the overhead caused by the resource elastic scheduler, we measured the time taken for execution of single wake up calls on an x86 Intel Core i7-6850K running Ubuntu 16.04 LTS and the total number of partial reconfiguration calls performed for schedulers across all the experiments. The results of this are shown in Figure B.6 and B.7, respectively. It was found that the computation overhead for a resource elastic scheduler is between 10× to 100× higher as compared to baseline schedulers. This is mostly due to our implementation, which enumerates all the possible module layouts and uses expensive rating functions. However, the total execution time is still below tens of microseconds which is negligible compared to the partial reconfiguration cost, which is in the range of milliseconds. Similarly, Figure B.7 shows that the resource elastic schedulers require about 12× to 100× more configuration calls than NS and a similar number of configuration calls to ACS. However, despite this, the performance of the resource elastic scheduler is much better with lower waiting time for the kernels, as shown in Figure B.8 and B.4. This is due to frequent partial reconfiguration for increasing resource utilisation which, in turn, improves performance.

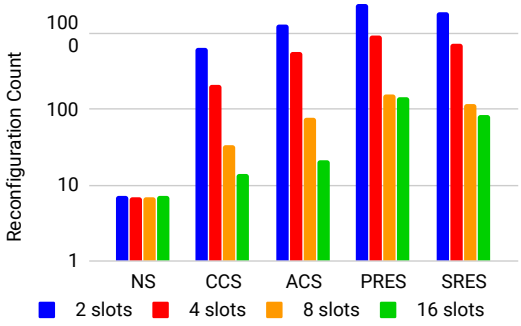


Figure B.7: Average number of reconfiguration calls performed by the schedulers for FPGAs with 2, 4, 8 and 16 slots.

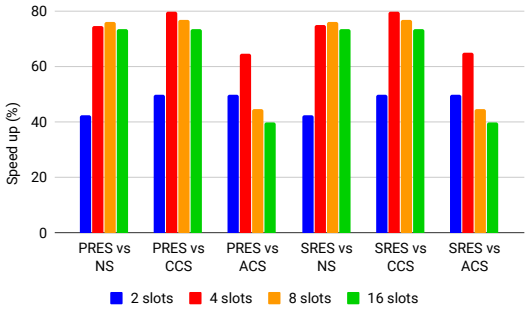


Figure B.8: Average speed-up of PRES and SRES compared to baseline schedulers.

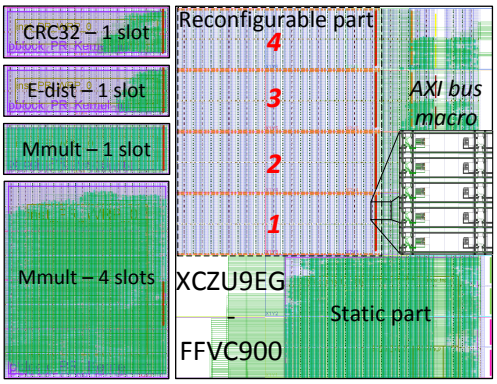


Figure B.9: Physical implementation of the base shell infrastructure and our in-house accelerators used in case study presented in Section B.3.

B.3 Application Case Study

In our case study, we deploy resource elastic scheduling on a recent Xilinx Zynq UltraScale+ platform for the same baseline schedulers and resource elastic schedulers as analysed in Section B.2. The platform used for the experiments is a TE8080 board featuring an XCZU9EG-FFVC900-2I-ES1 MPSoC device. This platform has the same characteristics as the ZCU102 board used in Chapter 4 with exception I/O layout which are not used in our experiments. The platform provides four reconfigurable regions (slots), as shown in Figure B.9, and each takes around 6.77 ms for partial reconfiguration.

We conducted the scheduling experiments on this platform with our in-house accelerators[1] for communication, arithmetic and machine learning (see Figure B.9): CRC32,

[1] At the time of experiments on this legacy runtime system, Spector benchmark used in Chapter 5

	SRES	PRES	ACS	CCS/NS
mmult wait time	28 ms	28 ms	28 ms	7 ms
CRC32 wait time	8 ms	8 ms	8 ms	7 ms
e-dist wait time	15 ms	15 ms	15 ms	14 ms
Total completion time	356 ms	404 ms	537 ms	1225 ms

Table B.1: Wait time of kernels and completion time of schedule for the case study. Where, A/B denotes that scheduling policies A and B provides the same results.

matrix multiplication (mmult), and Euclidean distance (e-dist) for k-means. The matrix multiplication kernel has two different physical implementations of slot sizes 1 and 4, where the 4 slot version offers 5× the performance of a 1 slot module due to better reuse of the data and a bigger work-group size. CRC32 and Euclidean distance kernels occupy 1-slot each.

The case study models a long-running kernel with matrix multiplication which needs to run 20480 work-items with an arrival time of zero. While CRC32 and Euclidean distance are modelled as short running kernels with 16384 work-items of low latency each and an arbitrarily chosen arrival time of 50 ms, such that it would interrupt the long-running kernel. The overall decisions taken by different schedulers are shown in Figure B.10 and their respective performance characteristics are captured in Table B.1. We can see that ACS provides the highest performance and utilisation among the baseline schedulers. However, with the ability to grow and shrink modules, SRES and PRES effectively maximise the utilisation and provides 33.7% and 24.7% better performance than ACS with similar waiting time, respectively. In this particular scenario, SRES outperforms PRES due to the lack of looking ahead in PRES, which leads to a greedy decision of accelerating matrix multiplication over CRC32 and hence, a higher total completion time.

was not available.

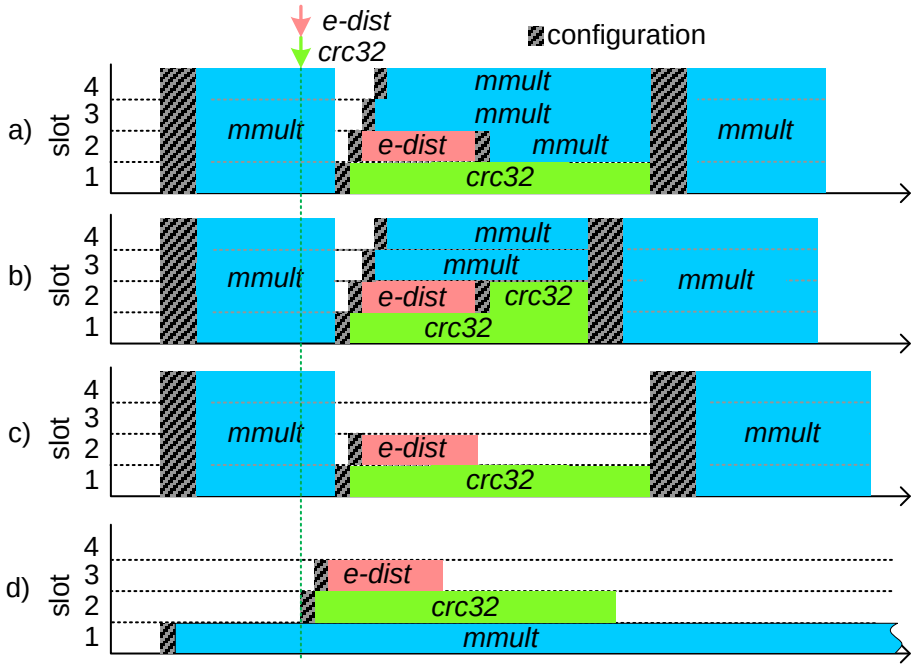


Figure B.10: Execution trace of a practical execution using the schedulers where a) is PRES, b) is SRES, c) is ACS, and d) is NS and CCS. Note, NS and CCS has the same trace in this example.

Bibliography

[1] N. Abbani et al. A Distributed Reconfigurable Active SSD Platform for Data Intensive Applications. In *International Conference on High Performance Computing and Communications*, pages 25–34, Sept 2011. doi:10.1109/HPCC.2011.14.

[2] J. Agron et al. Run-Time Services for Hybrid CPU/FPGA Systems on Chip. In *International Real-Time Systems Symposium (RTSS'06)*, Dec 2006. doi:10.1109/RTSS.2006.45.

[3] Altera. Nios® II Processor. URL: https://www.altera.com/products/processors/overview.html.

[4] J. Angermeier et al. Virtual Area Management: Multitasking on Dynamically Partially Reconfigurable Devices. In *International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, April 2010. doi:10.1109/IPDPSW.2010.5470754.

[5] M. Asiatici, N. George, K. Vipin, S. A. Fahmy, and P. Ienne. Virtualized Execution Runtime for FPGA Accelerators in the Cloud. *IEEE Access*, 5:1900–1910, 2017. doi:10.1109/ACCESS.2017.2661582.

[6] C. Beckhoff, D. Koch, and J. Torresen. GoAhead: A Partial Reconfiguration Framework. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2012.

[7] T. Beisel, T. Wiersema, C. Plessl, and A. Brinkmann. Programming and scheduling model for supporting heterogeneous accelerators in linux. In *Workshop on Computer Architecture and Operating System Co-design (CAOS)*, 2012.

[8] S. Biookaghazadeh, M. Zhao, and F. Ren. Are FPGAs Suitable for Edge Computing? In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*,

Boston, MA, July 2018. USENIX Association. URL: https://www.usenix.org/conference/hotedge18/presentation/biookaghazadeh.

[9] J. Blazewicz, J. Lenstra, and A. Kan. Scheduling subject to resource constraints: Classification and complexity. *Discrete Applied Mathematics*, 5(1):11 – 24, 1983. doi:https://doi.org/10.1016/0166-218X(83)90012-4.

[10] C. Bobda et al. The Erlangen Slot Machine: Increasing Flexibility in FPGA-based Reconfigurable Platforms. In *International Conference on Field-Programmable Technology (FPT)*, Dec 2005. doi:10.1109/FPT.2005.1568522.

[11] A. Bourge et al. Generating Efficient Context-Switch Capable Circuits Through Autonomous Design Flow. *ACM Trans. Reconfigurable Technol. Syst. (TRETS)*, Dec. 2016.

[12] A. Brant and G. G. F. Lemieux. ZUMA: An Open FPGA Overlay Architecture. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2012. doi:10.1109/FCCM.2012.25.

[13] S. Byma et al. FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2014. doi:10.1109/FCCM.2014.42.

[14] S. R. Chalamalasetti et al. An FPGA Memcached Appliance. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2013. doi:10.1145/2435264.2435306.

[15] H. Y. Cheah, S. A. Fahmy, and D. L. Maskell. iDEA: A DSP block based FPGA soft processor. In *International Conference on Field-Programmable Technology (FPT)*, 2012. doi:10.1109/FPT.2012.6412128.

[16] F. Chen et al. Enabling FPGAs in the Cloud. In *Proceedings of the ACM Conference on Computing Frontiers*, CF '14, 2014. doi:10.1145/2597917.2597929.

[17] Y. T. Chen et al. When Spark Meets FPGAs: A Case Study for Next-Generation DNA Sequencing Acceleration. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2016. doi:10.1109/FCCM.2016.18.

[18] C. H. Chou, A. Severance, A. D. Brant, Z. Liu, S. Sant, and G. G. Lemieux. VEGAS: Soft Vector Processor with Scratchpad Memory. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2011. doi:10.1145/1950413.1950420.

[19] K. Compton et al. Configuration Relocation and Defragmentation for Run-time Reconfigurable Computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10, 2002. doi:10.1109/TVLSI.2002.1043324.

[20] J. Cong, Z. Fang, M. Huang, L. Wang, and D. Wu. CPU-FPGA Coscheduling for Big Data Applications. *IEEE Design Test*, 35(1):16–22, 2018. doi:10.1109/MDAT.2017.2741459.

[21] J. Cong, H. Huang, C. Ma, B. Xiao, and P. Zhou. A Fully Pipelined and Dynamically Composable Architecture of CGRA. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2014. doi:10.1109/FCCM.2014.12.

[22] J. Coole and G. Stitt. Intermediate Fabrics: Virtual Architectures for Circuit Portability and Fast Placement and Routing. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 13–22, Oct 2010. doi:10.1145/1878961.1878966.

[23] J. Coole and G. Stitt. Fast, Flexible High-Level Synthesis from OpenCL using Reconfiguration Contexts. *IEEE Micro*, 2014. doi:10.1109/MM.2013.108.

[24] T. S. Czajkowski et al. From OpenCL to High-Performance Hardware on FPGAs. In *International Conference on Field Programmable Logic and Applications (FPL)*, 2012.

[25] K. Danne. Real-time Multitasking in Embedded Systems Based on Reconfigurable Hardware, 2006. Zugl.: Paderborn, Univ., Diss.

[26] O. Diessel et al. Dynamic Scheduling of Tasks on Partially Reconfigurable FPGAs. *IEE Proceedings - Computers and Digital Techniques*, 147, May 2000.

[27] K. Eguro. SIRC: An Extensible Reconfigurable Computing Communication API. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2010. doi:10.1109/FCCM.2010.29.

[28] E. El-Araby et al. Virtualizing and Sharing Reconfigurable Resources in High-Performance Reconfigurable Computing systems. In *International Workshop on High-Performance Reconfigurable Computing Technology and Applications*, Nov 2008. doi:10.1109/HPRCTA.2008.4745683.

[29] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and The End of Multicore Scaling. In *International Symposium on Computer Architecture (ISCA)*, 2011.

[30] S. A. Fahmy et al. Virtualized FPGA Accelerators for Efficient Cloud Computing. In *International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015. doi:10.1109/CloudCom.2015.60.

[31] S. A. Fahmy and K. Vipin. A Case for FPGA Accelerators in the Cloud. 2014.

[32] T. Feist. Vivado Design Suite. *White Paper*, 5:30, 2012.

[33] K. Fleming and M. Adler. *The LEAP FPGA Operating System*, pages 245–258. Springer International Publishing, Cham, 2016. doi:10.1007/978-3-319-26408-0_14.

[34] Q. Gautier et al. Spector: An OpenCL FPGA Benchmark Suite. In *International Conference on Field-Programmable Technology (FPT)*, 2016.

[35] Google. gRPC Framework, 2019. Accessed: 2019-12-19. URL: https://grpc.io/.

[36] V. Govindaraju et al. DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing. *IEEE Micro*, 32(5), 2012. doi:10.1109/MM.2012.51.

[37] M. A. D. Guzmán et al. Cooperative CPU, GPU, and FPGA heterogeneous execution with EngineCL. *The Journal of Supercomputing*, 75, 2019.

[38] F. Habibi et al. Resource-constrained project scheduling problem: review of past and recent developments. *Journal of Project Management*, 3(2), 2018.

[39] M. Happe, A. Traber, and A. Keller. Preemptive Hardware Multitasking in ReconOS. In *Applied Reconfigurable Computing (ARC)*, 2015.

[40] S. Hartmann. Packing Problems and Project Scheduling Models: an Integrating Perspective. *Journal of the Operational Research Society*, 51(9), 2000. doi:10.1057/palgrave.jors.2601011.

[41] S. Hartmann and D. Briskorn. A Survey of Variants and Extensions of the Resource-constrained Project Scheduling Problem. *European Journal of Operational Research*, 207(1), 2010. doi:10.1016/j.ejor.2009.11.005.

[42] J. L. Hennessy and D. A. Patterson. A New Golden Age for Computer Architecture. *Communications of the ACM*, 62(2):48–60, 2019.

[43] C. Huang et al. Dynamically Swappable Hardware Design in Partially Reconfigurable Systems. In *International Symposium on Circuits and Systems (ISCAS)*, 2007. doi:10.1109/ISCAS.2007.378620.

[44] M. Huang et al. Programming and Runtime Support to Blaze FPGA Accelerator Deployment at Datacenter Scale. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '16, 2016. doi:10.1145/2987550.2987569.

[45] S. Huang et al. Analysis and Modeling of Collaborative Execution Strategies for Heterogeneous CPU-FPGA Architectures. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE '19, 2019. doi:10.1145/3297663.3310305.

[46] A. Hugo et al. Composing Multiple StarPU Applications over Heterogeneous Machines: A Supervised Approach. In *International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPS)*, 2013.

[47] Intel FPGA. SDK for OpenCL. *Programming Guide. UG-OCL002*, 31, 2016.

[48] A. Iordache et al. High Performance in the Cloud with FPGA Groups. In *International Conference on Utility and Cloud Computing (UCC)*, Dec 2016.

[49] P. Jääskeläinen et al. pocl: A Performance-Portable OpenCL Implementation. *International Journal of Parallel Programming*, 43, 2015. doi:10.1007/s10766-014-0320-y.

[50] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner. RIFFA 2.1: A Reusable Integration Framework for FPGA Accelerators. *ACM Trans. Reconfigurable Technol. Syst. (TRETS)*, 8(4):22:1–22:23, Sept. 2015. doi:10.1145/2815631.

[51] A. K. Jain. *Architecture Centric Coarse-Grained FPGA Overlays*. PhD thesis, Nanyang Technological University, 2017.

[52] A. K. Jain et al. Virtualized Execution and Management of Hardware Tasks on a Hybrid ARM-FPGA Platform. *Journal of Signal Processing Systems*, 2014. doi:10.1007/s11265-014-0884-1.

[53] A. K. Jain, S. A. Fahmy, and D. L. Maskell. Efficient Overlay Architecture Based on DSP Blocks. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2015. doi:10.1109/FCCM.2015.15.

[54] R. Jain et al. Throughput Fairness Index: An Explanation. Technical report, Dept. of CIS, Ohio State University, 1999.

[55] G. Jo et al. OpenCL Framework for ARM Processors with NEON Support. In *Proceedings of the Workshop on Programming Models for SIMD/Vector Processing*, pages 33–40, 2014.

[56] V. Kathail et al. SDSoC: A Higher-level Programming Environment for Zynq SoC and Ultrascale+ MPSoC. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2016. doi:10.1145/2847263.2847284.

[57] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach. Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Oct. 2018.

[58] O. Knodel, P. R. Genssler, and R. G. Spallek. Migration of Long-running Tasks Between Reconfigurable Resources Using Virtualization. *SIGARCH Comput. Archit. News*, 44(4):56–61, Jan. 2017. doi:10.1145/3039902.3039913.

[59] O. Knodel, P. R. Genssler, and R. G. Spallek. Virtualizing Reconfigurable Hardware to Provide Scalability in Cloud Architectures. *Reconfigurable Architectures, Tools and Applications, RECATA 2017, ISBN: 978-1-61208-585*, 2017.

[60] D. Koch. *Architectures, Methods, and Tools for Distributed Run-time Reconfigurable FPGA-based Systems*. PhD thesis, University of Erlangen-Nuremberg, 2009.

[61] D. Koch et al. Efficient Hardware Checkpointing: Concepts, Overhead Analysis, and Implementation. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2007.

[62] D. Koch et al. Modeling and Synthesis of Hardware-Software Morphing. In *International Symposium on Circuits and Systems (ISCAS)*, May 2007. doi:10.1109/ISCAS.2007.378621.

[63] D. Koch et al. An Efficient FPGA Overlay for Portable Custom Instruction Set Extensions. In *2013 23rd International Conference on Field programmable Logic and Applications*, Sept 2013. doi:10.1109/FPL.2013.6645517.

[64] Y.-H. Lai et al. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019. doi:10.1145/3289602.3293910.

[65] A. Landy and G. Stitt. A Low-overhead Interconnect Architecture for Virtual Reconfigurable Fabrics. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2012. doi:10.1145/2380403.2380427.

[66] M. Lavasani, H. Angepat, and D. Chiou. An FPGA-based In-Line Accelerator for Memcached. *IEEE Computer Architecture Letters*, 13(2):57–60, July 2014. doi:10.1109/L-CA.2013.17.

[67] E. Lubbers and M. Platzner. ReconOS: An RTOS Supporting Hard-and Software Threads. In *International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2007. doi:10.1109/FPL.2007.4380686.

[68] P. Lysaght et al. Invited Paper: Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs. In *International Conference on Field Programmable Logic and Applications (FPL)*, 2006. doi:10.1109/FPL.2006.311188.

[69] R. L. Lysecky, K. Miller, F. Vahid, and K. A. Vissers. Firm-core Virtual FPGA for Just-in-time FPGA Compilation. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, page 271, 2005.

[70] L. Ma, F. B. Muslim, and L. Lavagno. High Performance and Low Power Monte Carlo Methods to Option Pricing Models via High Level Design and Synthesis. In *EMS*, Nov 2016. doi:10.1109/EMS.2016.036.

[71] K. Manev, A. Vaishnav, and D. Koch. Unexpected Diversity: Quantitative Memory Analysis for Zynq UltraScale+ Systems. In *International Conference on Field-Programmable Technology (FPT)*, 2019.

[72] K. Matas et al. Invited Tutorial: FPGA Hardware Security for Datacenters and Beyond. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2020.

[73] S. Mavridis et al. VineTalk: Simplifying Software Access and Sharing of FPGAs in Datacenters. In *International Conference on Field Programmable Logic and Applications (FPL)*, 2017. doi:10.23919/FPL.2017.8056788.

[74] I. Mavroidis et al. ECOSCALE: Reconfigurable Computing and Runtime System for Future Exascale Systems. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2016.

[75] T. Moorthy and S. Gopalakrishnan. IO and Data Management for Infrastructure As A Service FPGA Accelerators. *Journal of Cloud Computing*, 6(1):20, Aug 2017. doi:10.1186/s13677-017-0089-9.

[76] A. Morales-Villanueva and A. Gordon-Ross. Partial Region and Bitstream Cost Models for Hardware Multitasking on Partially Reconfigurable FPGAs. In *International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPS)*, 2015.

[77] A. Munshi. The OpenCL Specification. In *Hot Chips 21 Symposium (HCS)*. IEEE, 2009.

[78] F. B. Muslim, L. Ma, M. Roozmeh, and L. Lavagno. Efficient FPGA Implementation of OpenCL High-Performance Computing Applications via High-Level Synthesis. *IEEE Access*, 5, 2017. doi:10.1109/ACCESS.2017.2671881.

[79] J. Nunez-Yanez et al. Simultaneous Multiprocessing on a FPGA+CPU Heterogeneous System-On-Chip. In *Parallel Computing is Everywhere*, Advances in Parallel Computing, 2018. doi:10.3233/978-1-61499-843-3-677.

[80] J. Ouyang, S. Lin, W. Qi, Y. Wang, B. Yu, and S. Jiang. SDA: Software-defined Accelerator for Large-scale DNN Systems. In *IEEE Hot Chips Symposium (HCS)*, pages 1–23, Aug 2014. doi:10.1109/HOTCHIPS.2014.7478821.

[81] W. Peck et al. Hthreads: A Computational Model for Reconfigurable Devices. In *International Conference on Field Programmable Logic and Applications*, 2006. doi:10.1109/FPL.2006.311336.

[82] O. Pell, O. Mencer, K. H. Tsoi, and W. Luk. *Maximum Performance Computing with Dataflow Engines*, pages 747–774. Springer New York, New York, NY, 2013. doi:10.1007/978-1-4614-1791-0_25.

[83] K. D. Pham. *FPGA Virtualisation on Heterogeneous Computing Systems – Model, Tools and Systems*. PhD thesis, The University of Manchester, 2020.

[84] K. D. Pham, E. Horta, and D. Koch. BITMAN: A Tool and API for FPGA Bitstream Manipulations. In *Design, Automation Test in Europe Conference Exhibition (DATE)*. IEEE, 2017.

[85] K. D. Pham, K. Paraskevas, A. Vaishnav, A. Attwood, M. Vesper, and D. Koch. ZUCL 2.0: Virtualised Memory and Communication for ZYNQ UltraScale+ FPGAs. In *International Workshop on FPGAs for Software Programmers (FSP)*, pages 1–9, Sep 2019.

[86] K. D. Pham, A. Vaishnav, M. Vesper, and D. Koch. ZUCL: A ZYNQ UltraScale+ Framework for OpenCL HLS Applications. In *International Workshop on FPGAs for Software Programmers (FSP)*, pages 1–9, Aug 2018.

[87] C. Plessl and M. Platzner. Virtualization of Hardware-Introduction and Survey. In *ERSA*, 2004.

[88] A. Putnam et al. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *International Symposium on Computer Architecuture (ISCA)*, pages 13–24. IEEE Press, June 2014.

[89] K. Rupnow et al. Block, Drop or Roll(back): Alternative Preemption Methods for RH Multi-Tasking. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2009. doi:10.1109/FCCM.2009.30.

[90] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proc. of the ACM Euro. Conf. on Computer Syss.*, 2013. doi:10.1145/2465351.2465386.

[91] O. Segal, P. Colangelo, N. Nasiri, Z. Qian, and M. Margala. SparkCL: A Unified Programming Framework for Accelerators on Heterogeneous Clusters. *arXiv preprint arXiv:1505.01120*, 2015.

[92] A. W. Services. AWS EC2 FPGA Hardware and Software Development Kit., 2009. Accessed: 2017-12-04. URL: https://github.com/aws/aws-fpga.

[93] A. Severance and G. Lemieux. VENICE: A Compact Vector Processor for FPGA Applications. In *IEEE Hot Chips Symposium (HCS)*, 2011. doi:10.1109/HOTCHIPS.2011.7477515.

[94] A. Severance and G. G. F. Lemieux. Embedded supercomputing in FPGAs with the VectorBlox MXP Matrix Processor. In *CODES+ISSS*, 2013. doi:10.1109/CODES-ISSS.2013.6658993.

[95] Y. Shan et al. FPMR: MapReduce Framework on FPGA. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2010. doi:10.1145/1723112.1723129.

[96] S. Shukla, N. W. Bergmann, and J. Becker. QUKU: A FPGA Based Flexible Coarse Grain Architecture Design Paradigm using Process Networks. In *International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPS)*, 2007. doi:10.1109/IPDPS.2007.370382.

[97] S. Siegel et al. OpenCPI HDL Infrastructure Specification. *Tech. Rep.*, 2010.

[98] H. Simmler et al. Multitasking on FPGA Coprocessors. In *International Conference on Field Programmable Logic and Applications (FPL)*, 2000.

[99] H. K.-H. So and R. W. Brodersen. *BORPH: An Operating System for FPGA-Based Reconfigurable Computers*. PhD thesis, EECS Department, University of California, Berkeley, 2007.

[100] H. K.-H. So and C. Liu. *FPGA Overlays*. 2016. URL: https://doi.org/10.1007/978-3-319-26408-0_16, doi:10.1007/978-3-319-26408-0_16.

[101] A. Sprecher and A. Drexl. Multi-mode Resource-constrained Project Scheduling by a Simple, General and Powerful Sequencing Algorithm. *European Journal of Operational Research*, 1998.

[102] N. Tarafdar, N. Eskandari, T. Lin, and P. Chow. Designing for FPGAs in the Cloud. *IEEE Design Test*, 2017. doi:10.1109/MDAT.2017.2748393.

[103] N. Tarafdar et al. Enabling Flexible Network FPGA Clusters in a Heterogeneous Cloud Data Center. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, FPGA '17, 2017. doi:10.1145/3020078.3021742.

[104] D. Theodoropoulos, N. Alachiotis, and D. Pnevmatikatos. Multi-FPGA Evaluation Platform for Disaggregated Computing. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017. doi:10.1109/FCCM.2017.20.

[105] Y. Tian et al. Efficient OS Hardware Accelerators Preemption Management in FPGA. In *International Conference on Field-Programmable Technology (FPT)*, Dec 2019. doi:10.1109/ICFPT47387.2019.00069.

[106] M. Treaster. A Survey of Fault-Tolerance and Fault-Recovery Techniques in Parallel Systems. *CoRR*, abs/cs/0501002, 2005. URL: http://arxiv.org/abs/cs/0501002, arXiv:cs/0501002.

[107] K. H. Tsoi and W. Luk. Axel: A Heterogeneous Cluster with FPGAs and GPUs. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2010. doi:10.1145/1723112.1723134.

[108] A. Vaishnav, K. D. Pham, and D. Koch. A Survey on FPGA Virtualization. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 131–137, Aug 2018. doi:10.1109/FPL.2018.00031.

[109] A. Vaishnav, K. D. Pham, D. Koch, and J. Garside. Resource Elastic Virtualization for FPGAs Using OpenCL. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 111–117, Aug 2018. doi:10.1109/FPL.2018.00028.

[110] V. K. Vavilapalli et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the Annual Symposium on Cloud Computing*, SOCC '13, 2013. doi:10.1145/2523616.2523633.

[111] M. Vesper. *Dynamic Stream Processing Pipelines on FPGAs Examplified on the PostGreSQL DBMS*. PhD thesis, The University of Manchester, 2019.

[112] M. Vesper, D. Koch, and K. Pham. PCIeHLS: an OpenCL HLS framework. In *International Workshop on FPGAs for Software Programmers (FSP)*, pages 1–6, Sept 2017.

[113] K. Vipin et al. System-level FPGA Device Driver with High-level Synthesis Support. In *International Conference on Field-Programmable Technology (FPT)*, Dec 2013. doi:10.1109/FPT.2013.6718342.

[114] K. Vipin and S. A. Fahmy. DyRACT: A partial reconfiguration enabled accelerator and test platform. In *International Conference on Field Programmable Logic and Applications (FPL)*, 2014. doi:10.1109/FPL.2014.6927507.

[115] W. Wang et al. pvFPGA: Accessing an FPGA-based Hardware Accelerator in a Paravirtualized Environment. In *2013 International Conference on Hardware/-Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE, 2013.

[116] Z. Wang, S. Zhang, B. He, and W. Zhang. Melia: A MapReduce Framework on OpenCL-Based FPGAs. *IEEE Trans. Parallel Distrib. Syst.*, 27(12):3547–3560, Dec. 2016. doi:10.1109/TPDS.2016.2537805.

[117] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf. Enabling fpgas in hyperscale data centers. In *Intl Conf on Ubiquitous Intelligence and Computing and Intl Conf on Autonomic and Trusted Computing and Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, 2015. doi:10.1109/UIC-ATC-ScalCom-CBDCom-IoP.2015.199.

[118] J. Weerasinghe et al. Network-attached FPGAs for Data Center Applications. In *International Conference on Field-Programmable Technology (FPT)*, Dec 2016. doi:10.1109/FPT.2016.7929186.

[119] L. Wirbel. Xilinx SDAccel: A Unified Development Environment for Tomorrows Data Center. *The Linley Group Inc*, 2014.

[120] T. Xia, J. C. Prévotet, and F. Nouvel. Hypervisor Mechanisms to Manage FPGA Reconfigurable Accelerators. In *International Conference on Field-Programmable Technology (FPT)*, 2016. doi:10.1109/FPT.2016.7929187.

[121] S. Xiao et al. Transparent Accelerator Migration in a Virtualized GPU Environment. In *CCGRID*, 2012.

[122] Xilinx. MicroBlaze Soft Processor Core. URL: https://www.xilinx.com/products/design-tools/microblaze.html.

[123] Xilinx. Xilinx FFmpeg Library. Accessed: 4 Feb 2019. URL: https://github.com/Xilinx/FFmpeg-xma.

[124] Xilinx. *UG910 - Vivado Design Suite User Guide*. 2014.

[125] Xilinx. *Zynq UltraScale+ MPSoC Data Sheet: Overview*. 2017.

[126] Xilinx. Platform Overview — Xilinx Runtime 2019.1 documentation, 2019. URL: https://xilinx.github.io/XRT/master/html/platforms.html.

[127] Xilinx. PYNQ, 2019. URL: https://github.com/xilinx/pynq.

[128] Xilinx. UG1144 - PetaLinux Tools Documentation Reference Guide, 2019. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1144-petalinux-tools-reference-guide.pdf.

[129] Xilinx. UG909 - Vivado Design Suite User Guide: Partial Reconfiguration, June 2019. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug909-vivado-partial-reconfiguration.pdf.

[130] Xilinx. Vitis AI Library User Guide, Dec 2019. URL: https://www.xilinx.com/support/documentation/user_guides/ug1354-xilinx-ai-sdk.pdf.

[131] Xilinx. Xilinx OpenCV User Guide, Jun 2019. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1233-xilinx-opencv-user-guide.pdf.

[132] Xilinx. Xilinx SDAccel Examples, 2019. URL: https://github.com/Xilinx/SDAccel_Examples/.

[133] Xilinx. Edge AI Platform, 2020. URL: https://www.xilinx.com/products/design-tools/ai-inference/edge-ai-platform.html.

[134] J. H. C. Yeung et al. Map-Reduce as a Programming Model for Custom Computing Machines. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2008. doi:10.1109/FCCM.2008.19.

[135] P. Yiannacouras, J. G. Steffan, and J. Rose. VESPA: Portable, Scalable, and Flexible FPGA-based Vector Processors. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2008. doi:10.1145/1450095.1450107.

[136] M. Yoshimi et al. A Performance Evaluation of CUBE: One-Dimensional 512 FPGA Cluster. In *Reconfigurable Computing: Architectures, Tools and Applications*, 2010.

[137] J. Yu, G. Lemieux, and C. Eagleston. Vector Processing As a Soft-core CPU Accelerator. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2008. doi:10.1145/1344671.1344704.

[138] Y. Zha and J. Li. Virtualizing FPGAs in the Cloud. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, 2020. doi:10.1145/3373376.3378491.

[139] J. Zhang et al. The Feniks FPGA Operating System for Cloud Computing. In *Proceedings of the Asia-Pacific Workshop on Systems*, APSys '17, 2017. doi:10.1145/3124680.3124743.

[140] Q. Zhao et al. Enabling FPGA-as-a-Service in the Cloud with hCODE Platform. *IEICE Transactions on Information and Systems*, E101.D(2), 2018. doi:10.1587/transinf.2017RCP0004.