



Università degli Studi di Messina

Dipartimento di Ingegneria

Corso di Laurea Magistrale in
Ingegneria Elettronica per l'Industria

μ -VF: Enabling Virtualization of Embedded FPGAs

Relatore:

Chiar.mo Prof. Francesco Longo

Tesi di laurea di:

Vincenzo Bucaria

Correlatore:

Chiar.mo Prof. Francesco Restuccia

ANNO ACCADEMICO 2024/2025

Abstract

Despite growing interest in virtualization of Field-Programmable Gate Arrays (FPGAs), existing approaches predominantly target datacenter-class FPGAs, which heavily rely on external (powerful) servers for hypervisor execution and resource management. This significantly limits their suitability for edge environments where autonomy, energy efficiency, and direct low-latency access to physical Input/Output (I/O) are critical. To address this goal, this thesis work introduces μ -VF, a lightweight virtualization framework specifically designed to enable robust multi-tenancy on embedded FPGAs operating autonomously at the network edge. μ -VF embeds all virtualization logic entirely onboard the FPGA unit, eliminating the need for any off-chip infrastructure and thus significantly reducing overall system power consumption. Each tenant operates within a secure and isolated container on the on-chip Processing System (PS), coupled with exclusive access to a dedicated Programmable Logic (PL) region. Additionally, μ -VF fully virtualizes external General-Purpose Input/Output (GPIO) directly within the PL fabric, thus enabling independent, concurrent and latency-sensitive access to shared peripherals. We have implemented a prototype of μ -VF with a Zynq UltraScale+ ZCU102 board with PL operating at 100 MHz. Experimental results demonstrate that the hardware virtualization layer utilizes less than 10% of the FPGA's logic resources, with 85% available for tenant applications compared to 50% in prior work. Moreover, μ -VF adds 2.93% to Memory-Mapped I/O (MMIO) access latency compared to native execution for single-tenant operation, increasing to 6.5% with four concurrent tenants. Memory throughput measurements show 1.8% overhead for write operations and negligible impact on read operations, with aggregate throughput 17.1% higher than previous frameworks. Hardware-based GPIO remapping completes in 20 nanoseconds. μ -VF is fully open source to facilitate reproducibility and adoption by the scien-

tific community. The source code is available at the GitHub repository: <https://github.com/vincenzobucaria/u-VF-Enabling-Virtualization-of-Embedded-FPGAs>.

Contents

Abstract	I
Contents	III
Introduction	VI
1 Background	1
1.1 SoC-FPGA Architectures	1
1.2 Design Flow for SoC-FPGAs	3
1.2.1 Hardware Design Paradigms	3
1.2.2 From Description to Bitstream: Synthesis and Implementation	4
1.2.3 Software Design and HW/SW Interaction	6
1.3 Dynamic Partial Reconfiguration	7
1.4 Cloud Computing Paradigms	8
1.4.1 Cloud, Edge, and Fog Computing	9
1.4.2 Infrastructure as-a-Service (IaaS)	9
1.4.3 Sensing and Actuation as-a-Service (SAaaS)	10
1.5 Virtualization Concepts	10
1.5.1 FPGA Virtualization Concepts and Objectives	11
2 Architecture	12
2.1 Motivation and Challenges of Embedded Virtualization	12
2.2 System Overview and the vFPGA Concept	14
2.3 Hardware Stack	16

2.3.1	Lakes, Standard Interface, and ATLAS	17
2.3.2	Shore and Isolation Mechanisms	21
2.4	Processing System Virtualization and Software Stack	22
2.4.1	On-Device Hypervisor	24
2.4.2	Container-Based Tenant Execution Flow	25
3	Implementation Details	28
3.1	Experimental Platform and Tools	28
3.2	Hardware Stack Implementation	29
3.2.1	Basic Processing System Configuration	29
3.2.2	Creation of Lakes	29
3.2.3	Isolation Mechanisms via Decouplers and MMUs	34
3.2.4	ATLAS	36
3.2.5	Integration of the Hardware Stack	37
3.2.6	Generation of the Abstract Shell and Floorplanning	39
3.2.7	Adding user hardware design to the Lakes	42
3.3	Software Stack Implementation	45
3.3.1	Container startup and Hypervisor configuration file	45
3.3.2	APIs and Hypervisor overlay resources assignment	47
3.3.3	CMA Buffer management	53
3.3.4	MMIO management	59
4	Evaluation	62
4.1	Experimental Platform	62
4.2	Programmable Logic resource overhead	63
4.3	MMIO latency	65
4.4	Memory throughput	67
4.5	ATLAS evaluation	70
4.5.1	Request-to-Ready latency	71
5	Related Work	73

5.1	Approaches for Multi-Tenancy and Resource Optimization	73
5.2	Overlay Architectures for Bitstream Portability	75
5.3	Virtualization on SoC-FPGA Platforms	76
5.3.1	Analysis of FOS	76
5.3.2	Analysis of Ker-ONE	77
6	Conclusions	79
	References	A

Introduction

The proliferation of the Internet of things (IoT) is driving the growth of edge computing. Applications such as autonomous vehicles, industrial automation, and wearable healthcare demand tight latency bounds, predictable execution, and low power consumption – requirements that necessitate processing data close to its source. FPGAs align well with these demands as they provide deterministic execution, bounded latency and high energy efficiency.

The key issue is that traditional FPGA deployments rely on static resource allocation, where each workload is tied to a fixed hardware region and I/O configuration. This leads to severe resource underutilization, since a single task rarely uses the entire FPGA fabric. In addition, static mapping prevents run-time adaptation to failures or changing demands, as reconfiguration typically requires regenerating hardware designs, which are incompatible with real-time constraints. Dynamic resource sharing addresses these limitations by abstracting hardware boundaries [10, 3] and enabling multiple applications to efficiently share hardware resources. Existing work on FPGA virtualization [17, 15, 31, 15] targets datacenter scenarios with high-end PCIe-attached accelerators, assume abundant resources and rely on external orchestration, making them unsuitable for embedded contexts with strict area, power, and autonomy constraints. Crucially, they neglect I/O virtualization, which is fundamental in the IoT since devices need to interact with the physical world through sensors and actuators. Virtualization layers for PCIe-attached FPGAs are designed for external host-to-FPGA communication and introduce software and protocol overheads unsuited to embedded FPGAs with limited resources [29][11]. These approaches do not extend naturally to SoC-class FPGAs, where PS and PL are tightly integrated and communicate over internal buses, requiring virtualization strategies that support on-chip coordination and direct I/O access without external

orchestration. Moreover, datacenter-class FPGAs are fundamentally incompatible with edge scenarios, as they consume more power [25], depend on high-power (e.g., x64-hosted) orchestration over PCIe [26], and exceed the form factor and thermal envelope of mobile or battery-powered platforms.

Existing frameworks for embedded FPGAs [27, 28] typically virtualize only the PL, while the PS remains a centralized orchestrator that dispatches hardware tasks on behalf of tenants. As such, tenants submit PL requests to a shared PS, which then schedules and manages PL execution, similar to how a CPU offloads work to a GPU. While simple in nature, this paradigm breaks the fundamental FPGA paradigm where developers expect tight hardware-software co-design with direct control over both compute elements [5, 1, 16].

To address these fundamental issues, we present μ -VF, a lightweight virtualization framework that brings full multi-tenancy, dynamic I/O virtualization and autonomous orchestration to embedded FPGAs. Unlike prior work, μ -VF provides each tenant with a complete virtual FPGA (vFPGA) abstraction comprising both containerized PS access and private PL regions. To the best of our knowledge, this is the first framework to implement this abstraction entirely on-device, without external hypervisors or host systems. μ -VF enables a distributed Infrastructure-as-a-Service (IaaS) model where embedded FPGAs operate as autonomous compute nodes. Each device exposes isolated execution environments supporting diverse scenarios - from multi-user edge nodes in university labs to vehicles where mission-critical and guest workloads coexist securely. Tenants deploy application-specific accelerators and interact through tightly-coupled software, achieving hardware/software co-design with low-latency communication and high memory throughput. Importantly, I/O is dynamically assignable at runtime, while remaining directly accessible through asynchronous hardware paths. These performance results stem from μ -VF's ability to address core challenges in embedded FPGA virtualization. First, low-latency and isolated communication is achieved through per-tenant virtual device interfaces, exposed by a lightweight on-device hypervisor. This avoids context switches and host-side mediation while enabling direct access to control registers. Second, high memory throughput is sustained by virtualizing DDR access directly within the programmable logic, allowing tenant accelerators to interact with memory independently of the

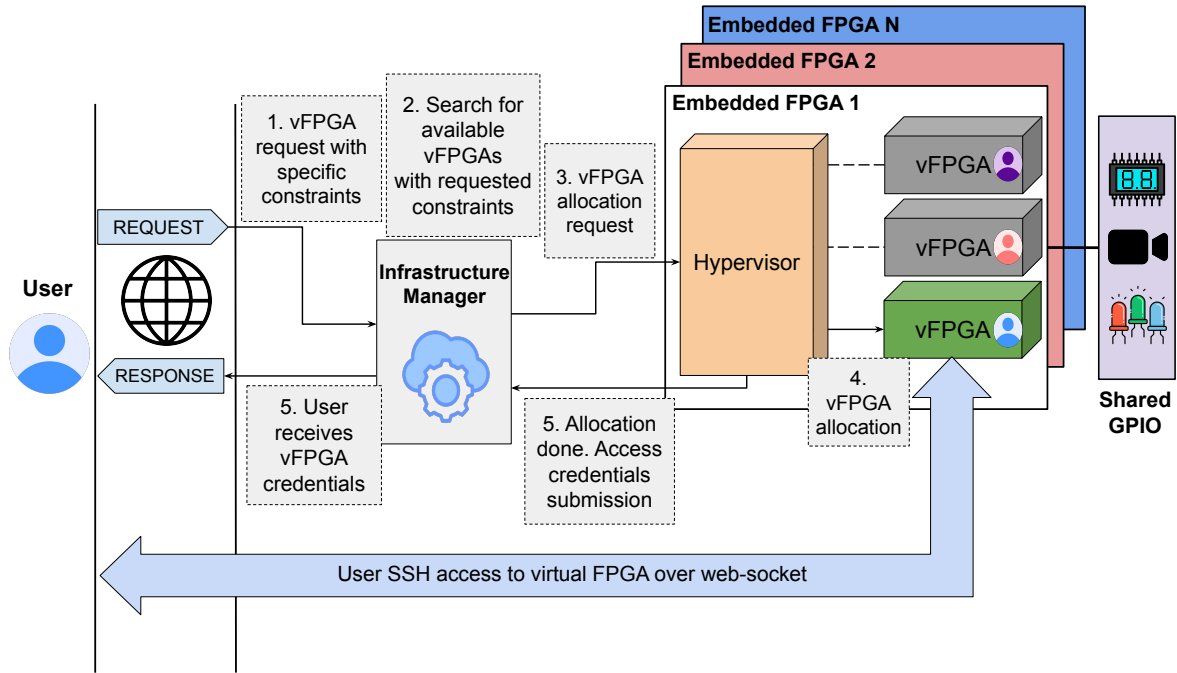


Figure 1: The μ -VF framework implements a Distributed Infrastructure-as-a-Service model where each embedded FPGA exposes multiple vFPGA that comprise both PS resources (containerized execution environments) and PL resources (isolated hardware regions). A central infrastructure manager orchestrates vFPGA allocation across the distributed fleet, while each device operates autonomously. Users connect directly to their assigned vFPGA and obtain dedicated PS/PL resources.

processing system. Third, asynchronous and parallel peripheral access is enabled through an I/O virtualization layer fully implemented in PL, which maps tenant logic to physical I/Os via virtual pins, bypassing the PS entirely. By sustaining sub 10% resource overhead and exposing up to 85% of the FPGA fabric to tenant logic, μ -VF improves over similar prior approaches [27], which, on the same ZCU102 platform, were limited to around 50% resource allocation due to virtualization overhead.

The key contributions of this thesis work can be summarized as follows:

- a fully on-device virtualization stack for embedded FPGAs, enabling secure multi-tenant execution with minimal resource overhead;
- dynamic GPIO virtualization supporting runtime remapping, time-sharing, and code portability across platforms;

- complete vFPGA abstraction integrating both PS and PL virtualization;
- support for edge-scale IaaS with embedded FPGAs as autonomous, remotely accessible compute nodes;
- an extensive performance evaluation on the ZCU102 board demonstrating: (i) less than 10% resource overhead with 85% of the fabric available for tenants, (ii) MMIO access latency increase of only 2,93% (single-tenant) to 6,5% (four concurrent tenants), (iii) memory throughput overhead below 1,8% with 10616 MB/s aggregate bandwidth, (iv) 20 ns GPIO remapping latency, and (v) comparison with existing work.
- The release of the entire μ -VF framework as a publicly available open-source implementation [36].

CHAPTER

1 | Background

1.1 SoC-FPGA Architectures

An FPGA, an acronym for Field-Programmable Gate Array, is an integrated device that can be reconfigured on-the-field an ideally unlimited number of times after its production. Traditionally, such devices consist of a matrix of reprogrammable logic blocks (in jargon, CLBs), whose behavior and interconnections can be defined through on-the-field reconfiguration, allowing the user, within the device's area limits, to implement any logic circuit.

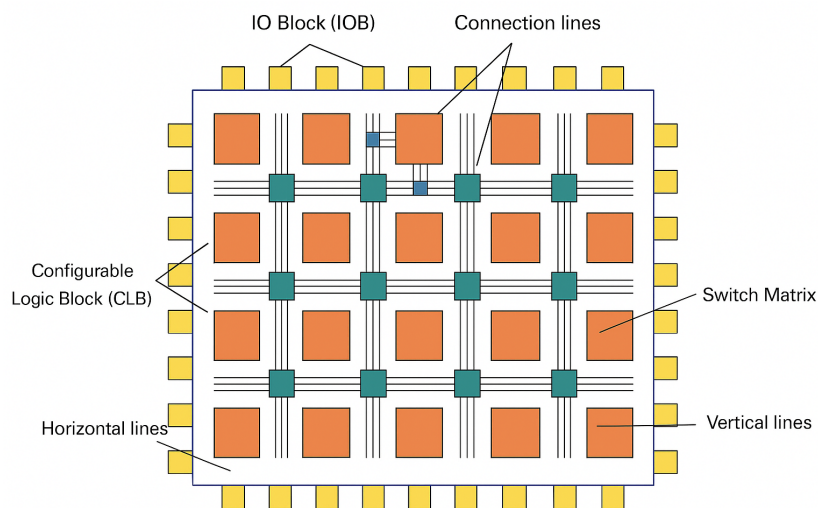


Figure 1.1
Generic "island-style" FPGA architecture

Figure 1.1 provides an overview of a generic FPGA architecture. Although low-level architectural details are proprietary to manufacturers and vary depending on the family and model, the main elements are discussed here. The CLBs provide the basic logic and memory functionalities, allowing both the execution of combinational functions and the implementation of sequential functions. Each CLB is, in turn, divided into multiple slices, which are elementary logic blocks that constitute the minimum programmable computing unit. Within each slice, there are typically look-up tables (LUTs), flip-flops (FFs), and multiplexers, which are the fundamental resources needed to define arbitrary logic functions, store states, and route signals. The routing infrastructure allows the interconnection of the various logic and functional blocks and is organized as a horizontal and vertical grid of connection segments, interconnected via Switch Matrices controlled by bits of the configuration memory, similarly to the CLBs, to establish the desired interconnections. The routing infrastructure extends not only to the interconnection between the various PLs but also to the I/O blocks, which allow the FPGA to communicate with external devices, and to hard blocks, which are blocks with pre-built functionalities integrated directly into the silicon. Among the most common hard blocks in modern FPGA architectures are DSPs, BRAM memories, and DAC/ADC converters.

The evolution of this architecture has led to the birth of modern SoC-FPGAs, which integrate the traditional PL, just described, with a processor system, in jargon called PS, on a single chip. Platforms like the AMD Zynq and UltraScale+, used in this thesis work, are an example of this. Typically, the PS is not a simple processor but a complete SoC in its own right. It includes, in addition to multi-core processors typically based on ARM architecture, memory controllers, standard communication interfaces (like Ethernet, USB, UART), or integrated GPUs.

In these devices, the PS, typically based on ARM architecture, is capable of running complex operating systems (e.g., Linux) and is tightly coupled to the FPGA fabric via high-performance internal communication buses, such as the AXI standard.

This tight integration represents the decisive advantage of SoC-FPGAs for embedded systems. Unlike architectures for data centers, where a host processor communicates with the FPGA via a slower external bus like PCIe, the PS-PL coupling enables a true hardware/software

co-design paradigm. Software applications running on the PS can manage, control, and exchange data with custom hardware accelerators in the PL with very low latencies, combining the flexibility and ease of software programming with the performance of specialized hardware.

1.2 Design Flow for SoC-FPGAs

The realization of an embedded application on an SoC-FPGA requires a multi-level design flow, which involves both the hardware design and the software design parts. Unlike traditional FPGAs, the integration between the Processing System (PS) and Programmable Logic (PL) introduces the need to coordinate the development of customized hardware components with that of the software that uses them, which is typically run on an operating system hosted by the PS.

The co-design paradigm on SoC-FPGAs is based on a tight interaction between the software running on the PS and the custom hardware on the PL. In the simplest model, the software running on the PS acts as the orchestrator, while the hardware design on the PL operates as a specialized accelerator. This interaction typically manifests on two distinct channels: a control plane, through which the software configures, starts, and monitors the accelerator by writing to low-latency registers, and a data plane, used to transfer massive data flows from system memory (DDR) to the accelerator for processing. However, describing the PS only as an orchestrator could be reductive, especially on platforms like the AMD UltraScale+ ZCU102. Devices of this kind enable a more advanced paradigm of heterogeneous computing, where the cores of the PS and the fabric work in parallel on significant workloads.

This dual interaction is at the heart of the design flow: the hardware must be designed to expose these interfaces, while the software must be developed to use them efficiently.

1.2.1 Hardware Design Paradigms

The hardware design process has the ultimate goal of generating a configuration file (bitstream), which is necessary to configure the logic and interconnections of the FPGA. Traditionally, the starting point for bitstream generation systems is the description of the desired hardware using

hardware description languages (HDLs), such as VHDL, Verilog, or SystemVerilog. More recently, higher-level design approaches have become popular, such as high-level synthesis (HLS) or model-based design with MATLAB/Simulink. These approaches do not replace the HDL paradigm but are placed at a higher level of abstraction: the model or high-level code is automatically translated into HDL, which remains the actual input for synthesis and implementation tools. In addition to the approaches just discussed, the use of graphical design environments provided by the tools themselves, such as Vivado's block designs, is also very common today. In this case, the design is done by composing known functional blocks as IP cores. An IP core is a reusable logic unit that encapsulates a specific functionality, such as a processor, a memory controller, or a communication interface. These blocks can be provided directly by the FPGA vendor in an already optimized and ready-to-use form or developed by the users themselves to create custom logic. It is important to note that modern projects rarely rely on a single approach but rather adopt a hybrid model, as in the case of this thesis work, where all three approaches were combined.

In addition to allowing this hybrid approach, modern EDA suites like Vivado provide a series of high-level functionalities that drastically simplify system integration. For example, they offer the possibility of encapsulating custom logic within standard bus interfaces (IP wrapping), provide tools like the Address Editor to assign and manage memory address ranges for each peripheral (MMIO), and finally allow the designer to guide the entire implementation process through targeted directives, controllable via both scripts (e.g., TCL) and a graphical interface.

1.2.2 From Description to Bitstream: Synthesis and Implementation

Regardless of the starting paradigm, the hardware design is processed by a suite of EDA tools to be transformed into the bitstream. For the Xilinx/AMD platforms used in this thesis work, this role is played by the Vivado Design Suite.

Figure 1.2 illustrates the development flow which, starting from the design paradigms discussed, ends with the generation of the bitstream for the FPGA configuration. This process is divided into three macro-phases: synthesis, implementation, and bitstream generation.

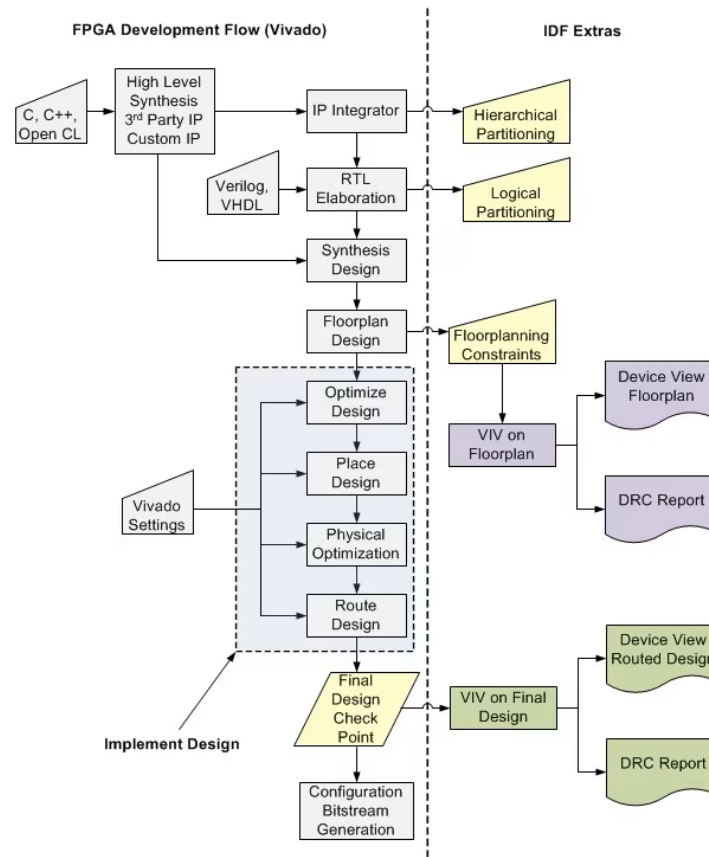


Figure 1.2
Bitstream generation flow

The synthesis phase aims to translate the abstract description of the hardware into a structural logical representation. The process begins with RTL elaboration, where the different project sources are analyzed and combined into a first RTL netlist. Subsequently, the Synthesis Design phase processes this netlist to generate a logical representation composed of generic components that can be mapped onto the specific FPGA technology, such as Look-Up Tables (LUTs), flip-flops, BRAMs, and DSPs.

The implementation phase aims to map the netlist generated by the synthesis onto the physical resources of the FPGA. This process is divided into multiple steps, the first of which is a high-level strategic phase: floorplanning. During the Floorplanning phase, the arrangement of the hardware design's macro-blocks on the FPGA area is outlined. Complex designs are typically structured as a composition of multiple modules. As will be seen in detail later in the discussion, through specific constraints (floorplanning constraints), it is possible to assign

well-defined physical area portions of the FPGA to specific modules, such as those that are part of a complex block design. After floorplanning, the implementation flow continues with a sequence of steps that include logic optimization, placement, and routing. In these phases, the generic netlist is concretely mapped onto the primitive resources of the target FPGA. These phases, particularly Place and Route, represent the most computationally intensive step of the entire development flow, requiring processing times that can extend for hours. This intensity derives from the fact that the process is, in essence, a vast combinatorial optimization problem. The outcome is not unique but is the result of a complex balance guided by the constraints and strategies imposed by the designer. The tool must explore a solution space of astronomical dimensions to find a configuration that satisfies a set of often competing objectives, such as minimizing the occupied area, reducing power consumption, and, above all, meeting stringent timing constraints.

The development flow culminates in the Bitstream Generation phase. In this final step, the physical mapping of the design, produced by the implementation tool, is translated into a binary configuration file. This file, the bitstream, contains the complete set of instructions necessary to program the primitive elements of the FPGA (described in Subsection 1.1) so that they replicate exactly the logical and physical structure defined by the designer.

1.2.3 Software Design and HW/SW Interaction

Based on what was said previously, the hardware-software interaction play an extremely important role on SoC-FPGAs. Depending on the target application, two different paradigms can be adopted for the development of the software running on the PS.

In the bare-metal execution approach, the software is executed directly on the processor hardware without the intermediation of an operating system. This approach guarantees maximum performance and a completely deterministic temporal behavior, but at the cost of greater development complexity as every resource must be managed by the designer at a low level.

In the operating system execution paradigm, the software is executed as an application within the OS. The performance overhead and lower determinism due to the operating sys-

tem's intermediation represent a widely accepted compromise in exchange for the enormous advantage of having an operating system on an embedded device. In fact, it enables the use of complex network stacks, robust file systems, and software ecosystems.

Given the objective of this thesis work, which is to virtualize embedded SoC-FPGAs to provide multiple users with virtual SoC-FPGA instances, the operating system execution paradigm was chosen. The choice fell specifically on the PYNQ framework [38], which is based on a standard Ubuntu distribution, for two fundamental reasons that guided its adoption.

First, PYNQ is based on a standard Ubuntu distribution, ensuring access to a vast software ecosystem. This allowed for the integration of enabling technologies like Docker, which is fundamental for containerization and the isolation of tenants at the software level. Second, and crucially important, PYNQ provides a powerful set of APIs that lower the barrier to entry for using the FPGA. The target user of a Virtual SoC-FPGA may not be an expert in embedded systems but a software programmer who wants to leverage hardware acceleration. The intended use model of the framework allows such a user to create accelerators with standard interfaces (e.g., AXI) through high-level tools like HLS. At this point, PYNQ abstracts the entire complexity of deployment: starting from the handoff file (.hwh) generated by Vivado, the framework automatically manages the Device Tree configuration and the loading of the bitstream (including the partial one, as discussed later). The end user is exposed to simple Python APIs to interact with their design, allowing them to: write and read from their peripherals with single instructions, allocate physically contiguous memory buffers for transfers to DDR memory, and, essentially, manage the life cycle of their accelerators in a software-like manner.

1.3 Dynamic Partial Reconfiguration

The set of processes described in section 1.2 defines the static reconfiguration flow, where a single bitstream is generated to configure the entire FPGA. Although powerful, this model has a fundamental limitation: any modification to the hardware design requires interrupting the activities in the programmable logic to load a new bitstream. This rigidity is incompatible

with multi-tenant environments, where each user must be able to configure their own region of the FPGA dynamically without interfering with the workloads on the regions assigned to other users.

This limitation can be overcome by using the paradigm of DPR, which allows specific portions of the PL to be dynamically modified while the rest of the system continues to operate without interruption. In AMD/Xilinx platforms, this technology is known as Dynamic Function eXchange (DFX) [34].

The DFX design flow is based on partitioning the design into an immutable static region and one or more reconfigurable partitions. The result of this flow is not a single file but a coordinated set: a main bitstream for the initial configuration of the static part and multiple partial bitstreams, one for each hardware module that can be loaded dynamically, provided that the interface between the static region and the modules remains unchanged. The DFX design flow, which we will see in more detail later, uses specific constructs provided by modern EDA tools. At a logical level, Vivado introduces Block Design Containers: a hierarchical block within a design can be marked as a "container," allowing the designer to define different alternative implementations for the same interface.

This logical partitioning must then be mapped onto the physical structure of the chip. This is done through manual floorplanning, with which the designer explicitly assigns each reconfigurable partition to a well-defined and isolated physical area of the FPGA.

1.4 Cloud Computing Paradigms

The cloud computing paradigm is now a pervasive component of everyday life, manifesting in streaming services, web applications, and access to remote computing resources. Nevertheless, a formal and unambiguous definition of the term remains elusive. This ambiguity mainly stems from the vast heterogeneity of service models and application scenarios that cloud computing itself enables [20].

To classify the different cloud computing models, at least two multiple dimensions can be used, but the most common ones are based on the physical arrangement of the provided

resources (centralized cloud, edge, fog) and the service model, which defines the level of abstraction made available to the user.

1.4.1 Cloud, Edge, and Fog Computing

The traditional cloud computing model is based on large centralized data centers. Although they have virtually unlimited computational capacity, the geographical distance from end-users introduces significant latencies for latency-sensitive applications, such as real-time security systems.

For latency-sensitive applications or those that require local processing, leveraging resources available only in certain locations (e.g., data, sensors, or actuators), as in IoT systems or cyber-physical contexts, complementary paradigms have spread:

- **Edge computing**, which brings computation closer to the data source, typically at the gateway or smart device level;
- **Fog computing**, which is located between the cloud and the edge, distributing resources to intermediate nodes to balance capacity and proximity.

These models are often integrated into hybrid architectures, where the cloud remains the point of global aggregation and orchestration, while edge and fog provide low-latency distributed capabilities.

1.4.2 Infrastructure as-a-Service (IaaS)

The Infrastructure as a Service (IaaS) model [22] virtualizes computing resources, allowing multiple tenants to run their workloads on a shared infrastructure while ensuring strict mutual isolation.

A similar analogy is found in the context of FPGAs: the heterogeneous hardware is abstracted behind a unified interface, so that each user can access, on demand, hardware acceleration spaces (reconfigurable logic regions) or traditional computing resources, such as the CPUs integrated into SoC-FPGAs. This space can be derived from the entire SoC-FPGA or a portion of it, depending on the sharing policies adopted. Ideally, the user should not care

whether they are operating on an entire FPGA or a fraction of it: what they receive is a virtual instance consistent with the interfaces and tools provided by the system. However, since it is low-level hardware design, the user must still be aware of certain physical constraints, such as the actual available area, the maximum operating frequencies, or memory limits, so that their design is synthesizable and functional.

1.4.3 Sensing and Actuation as-a-Service (SAaaS)

An extension of the IaaS model is Sensing and Actuation-as-a-Service (SAaaS) [8]. This model applies virtualization also to physical devices, treating sensors and actuators as requestable and manageable resources. SAaaS is crucial for scenarios where applications require not only computing capacity but also direct, low-latency interaction with the physical world, such as in environmental monitoring or robotic control.

1.5 Virtualization Concepts

The term virtualization was introduced in the field of computer science to describe the ability of a system to provide an abstraction of the underlying hardware resources. In one of the first and most well-known definitions, Popek and Goldberg describe a virtual machine as "an efficient, isolated duplicate of the real machine" [19]. More recently, virtualization has been defined as "the abstraction of computing resources that allows multiple operating systems and applications to share the same physical resources" [33], emphasizing the key role of abstraction for the concurrent and shared use of resources.

Generally, these processes are managed by a software layer called a hypervisor, which creates and manages one or more virtual environments (such as Virtual Machines) on a single physical host. A lighter form of virtualization is containerization, which operates at the operating system level to create isolated application instances.

Regardless of the technique, the primary objective is to ensure isolation: each virtual instance operates as an "efficient and isolated copy of the real machine," allowing multiple operating systems and applications to securely share the same physical resources.

1.5.1 FPGA Virtualization Concepts and Objectives

In the context of FPGAs, the term virtualization does not have a single meaning but encompasses different objectives and techniques. A first interpretation [9] concerns the adoption of principles borrowed from operating systems, such as partitioning and segmentation, which allow for the subdivision and management of the device's resources as independent entities.

A second thread is based on the concept of overlay architecture [24], which is the definition of an abstract logical layer placed above the reconfigurable fabric. This approach aims to decouple designs from the physical hardware, making bitstreams compatible between different FPGA families and thus facilitating the portability and uniform reprogramming of projects.

Other perspectives include techniques like temporal partitioning, virtualized execution, and mapping onto abstract virtual machines, which aim to make the best use of available resources and provide standard interfaces to developers.

In the current cloud and edge computing scenario, FPGA virtualization is mainly associated with objectives such as:

- **Abstraction:** hiding hardware details and offering simple and familiar interfaces to developers;
- **Multi-tenancy:** allowing the efficient sharing of resources among multiple users and applications;
- **Resource management:** ensuring transparent and balanced allocation of reconfigurable resources;
- **Isolation:** ensuring separation and protection between different users, in terms of performance and data security.

To achieve these goals, many frameworks adopt a model that partitions the system into a static shell, which provides basic services (e.g., memory access), and one or more dynamic regions, which host the users' reconfigurable logic. While several virtualization frameworks have been proposed in the literature, applicability to resource-constrained embedded platforms remains limited, as discussed further in this work.

CHAPTER

2 | Architecture

This chapter describes the architecture of the μ -VF framework, illustrating the abstraction model it offers to its users. The discussion begins by outlining the motivations behind embedded virtualization and its main challenges, then proceeds with an overview of the system and a detailed analysis of the hardware and software components that constitute its foundation.

2.1 Motivation and Challenges of Embedded Virtualization

In Chapter 1, we briefly discussed how FPGAs are devices with high flexibility, derived from the possibility of instantiating ad-hoc hardware accelerators for specific applications. However, the general level of flexibility is attenuated by the traditional static resource allocation method, based on a static allocation. In this resource allocation model, a single user generally reserves the entire device regardless of the area they actually need. It is certainly possible for multiple users to coexist on the same FPGA, but this requires accurate, conscious, and coordinated design upstream, which can only be achieved if all users are aware that they must share the same device. In practice, this represents a significant limitation: for applications that are not particularly complex, in fact, reserving an entire FPGA for a single user leads to an obvious waste of computational resources. Furthermore, the static assignment of I/O resources reduces the possibility of concurrent access by multiple users and forces the generation of a new bitstream every time the configuration needs to be modified, for example, following

a malfunction of the currently assigned pins. However, regenerating a bitstream is not an immediate operation: even a simple remapping to new physical pins requires the entire place route process, with execution times that are often not negligible and certainly not instantaneous. This is incompatible with scenarios where it is necessary to dynamically reconfigure I/O resources, such as in fault-tolerant systems or applications that require real-time adaptation.

The advent of FPGA devices in the infrastructures of important cloud providers, such as Amazon, Alibaba, and Azure, where each user is generally dedicated an entire FPGA, has attracted the attention of the scientific community to the problem of optimizing the use of hardware resources and enabling multi-tenancy at the data center FPGA level. Although the sharing of a single FPGA among multiple tenants is not implemented in current commercial services, this scenario has stimulated the development of frameworks and architectures in the literature aimed at maximizing the use of resources with multi-tenant environments. So, while the interest has focused on data center FPGAs, with the goal of instantiating more hardware accelerators for computationally complex tasks, little attention has been paid to embedded FPGA boards, such as SoC-FPGAs, unconstrained by data center architectures.

Virtualizing SoC-FPGAs involves solving specific challenges compared to the data center counterpart. First, the area offered by these devices is significantly smaller. To give an order of magnitude, the AMD ZCU102 used in this thesis has about 274,000 Look-Up Tables (LUTs), while a data center entry-level one like the AMD Alveo U50 has about 780,000 LUTs. The difference becomes even more marked with lower-end but very widespread SoC-FPGAs, such as the ZedBoard, which has only 53,000 LUTs. The smaller amount of available area implies that all hardware virtualization mechanisms must be optimized to minimize the overhead in terms of occupied logic. Second, a characteristic element of embedded platforms is the need to interact with the outside world through the GPIO subsystem, whose virtualization represents an essential requirement to ensure the isolation and sharing of peripherals. A further challenge is represented by the heterogeneity of the platforms: different FPGA boards have different pin layouts and amounts of resources, making hardware designs not directly portable. An effective virtualization framework must therefore provide a level of abstraction that hides these specificities. Furthermore, while in data center scenarios a large part of the virtualization

services can be delegated to the host, typically a general-purpose x86 machine, in embedded applications the FPGA must operate in a stand-alone mode. In such contexts, in fact, it is not possible to resort to an external hypervisor, both for energy and availability constraints, and because IoT and edge computing applications require lightweight and self-contained solutions. On this last aspect, SoC-FPGAs offer the possibility of using and extending operating systems, which allows for the integration of resource management and virtualization mechanisms directly on the device. However, the operating systems used must have a low impact on the overall overhead, both in terms of CPU usage and latency, so as not to compromise the performance of the tenants and respect the typical constraints of embedded and IoT applications. A further aspect, peculiar to SoC-FPGAs, is the need to virtualize not only the fabric, but also the integrated processing system, which plays the role of host and access point to the device's resources. Virtualizing the PS therefore means allowing multiple tenants to obtain an isolated instance that is strongly coupled with its own fabric partition, enabling hardware/software co-design scenarios where reconfigurable logic and software cooperate as if each user had a dedicated device. This introduces further challenges: on one hand, the need to guarantee isolation and fairness in CPU computing resources, and on the other, the minimization of the overhead introduced by scheduling or containerization mechanisms, so as not to compromise the latency and energy efficiency constraints typical of embedded and IoT applications.

2.2 System Overview and the vFPGA Concept

As anticipated in the introductory chapter, the goal of μ -VF is to enable virtualization and multi-tenancy on SoC-FPGAs in a completely autonomous way, without depending on hypervisors running on external hosts. More precisely, μ -VF allows integrating SoC-FPGAs into IaaS, offering users an abstraction similar to that of virtual machines in traditional servers: the virtual-FPGA (vFPGA) instance.

A vFPGA represents a complete and isolated portion of the physical device, guaranteeing each user:

- A private area partition in the FPGA fabric (Programmable Logic) intended for accelerators or custom hardware designs;
- An isolated software execution environment on the Processing System, tightly coupled to the hardware partition;
- Virtualized access to physical peripherals, through a set of virtual I/O pins that can be dynamically mapped onto the real GPIO of the host board.

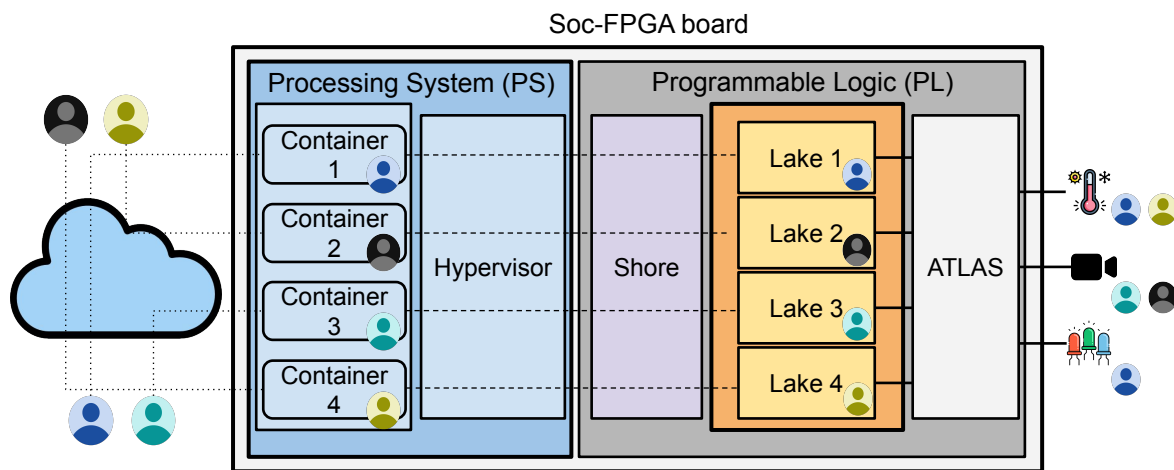


Figure 2.1: Overview of the μ -VF architecture. Each tenant operates within an isolated software container running on the Processing System (PS), with access to a private Lake in the programmable logic (PL). The Shore manages access to shared resources such as DDR memory and provides the communication interface between Lakes and the PS. A hardware-based virtualization layer, ATLAS, dynamically maps virtual I/O pins to physical devices. The entire architecture is orchestrated by a local hypervisor and supports multi-tenant execution without requiring external coordination.

The μ -VF architecture implements the vFPGA concept through a unified abstraction that covers both the Programmable Logic (PL) and the Processing System (PS), as illustrated in Figure 2.1.

The system is divided into a hardware stack on the PL and a software stack on the PS, which operate in close synergy. The hardware stack adopts a conceptual partitioning model where the "sea of resources" of the PL is divided into two types of components. The first is the Shore, a static and shared infrastructure that mediates access to common services (memory, reconfiguration, interaction with the PS). The second component consists of multiple Lakes,

which are isolated reconfigurable regions that are dynamically assigned to tenants for the deployment of their hardware designs.

Still within the hardware stack, the ATLAS (Abstract Tenant-Led Access to Signals) virtualization layer enables direct, low-latency connection between tenant logic and physical I/O devices, without routing communications through the Processing System.

In parallel, on the Processing System (PS), a set of lightweight software containers provides each tenant with an isolated execution environment for their applications. The entire system is orchestrated by a minimal on-device hypervisor, which manages resource allocation, coordinates partial reconfiguration, and mediates all interactions through APIs.

2.3 Hardware Stack

The μ -VF hardware stack was designed to balance two critical requirements for embedded systems: minimal resource impact and maximum performance. To achieve this balance, a hybrid approach was adopted: time-critical operations, such as GPIO signal routing and memory access control, are implemented directly in hardware on the PL; conversely, management and decision-making functions, such as resource allocation and policy application, are delegated to the hypervisor running on the PS, saving precious area on the PL.

The hardware stack includes two main categories of components:

- The static components (Shore and ATLAS), which provide the fundamental virtualization services, are synthesized and instantiated on the FPGA only once at system startup and persist unchanged during the entire operation of the device.
- The dynamic components (Lakes), which can be allocated, programmed, and de-allocated independently at runtime using partial reconfiguration. When a user requests to instantiate their own accelerator, only their specific Lake is configured via the FPGA's configuration port (ICAP), leaving the static infrastructure and the other Lakes fully operational.

2.3.1 Lakes, Standard Interface, and ATLAS

As already mentioned, each tenant is assigned a Lake, a partially reconfigurable portion of the FPGA fabric at their complete disposal, with resources such as LUTs, flip-flops, and BRAM. Within this region, the user is free to implement their own hardware designs, using both description languages (HDL) and high-level synthesis tools (HLS), provided that the design respects a standardized top-level interface.

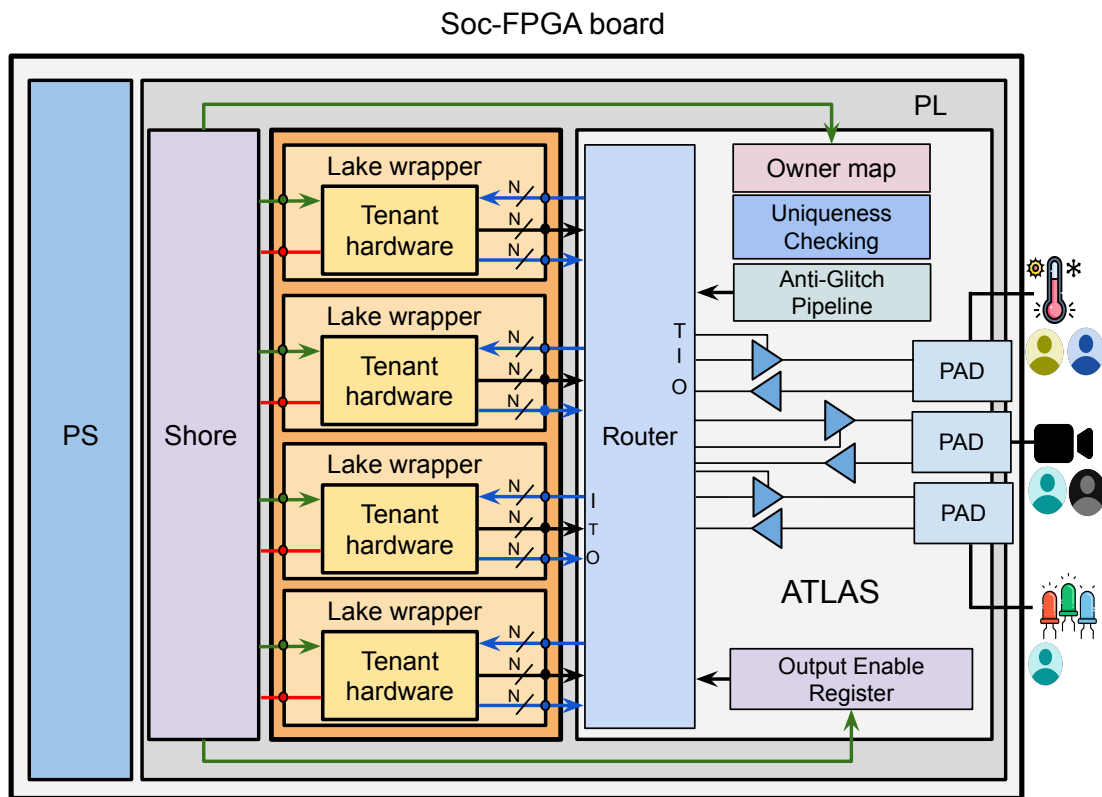


Figure 2.2: ATLAS GPIO virtualization architecture in μ -VF. Tenant hardware within Lake wrappers exposes N tri-state virtual pins (I/O/T signals) that ATLAS routes to physical pads via a configurable router. The Owner Map and Anti-Glitch Pipeline ensure safe dynamic remapping, while the Output Enable Register enforces access control. **Legend.** Green: AXI-Lite bus; Red: AXI-Full bus; Blue arrows: virtual GPIO signals (direction indicates input/output); Black arrows: internal signal connections; Black lines: physical GPIO connections.

Standardized Interface

As illustrated in Figure 2.2, each user's design is encapsulated in a Lake wrapper, which exposes a fixed interface to the static infrastructure (the Shore). This wrapper abstracts the complexity of the virtualization layer and, as will be seen in the implementation chapter, its immutable interface is a fundamental requirement to enable Partial Reconfiguration.

The Lake interface was designed to reflect the typical requirements of embedded systems. Specifically, it must guarantee each partition: (i) coordination with the software processes running on the PS, (ii) access to shared memory subsystems, such as the DDR, (iii) access to external peripherals, such as sensors and actuators, via the GPIO.

To support these requirements, the interface exposes three communication channels:

- A control channel, implemented on a 32-bit bitwidth AXI-Lite bus, which enables memory-mapped access to the user's accelerator registers by the PS. The choice of AXI-Lite with a reduced bitwidth is motivated by its lightness.
- A data channel based on a 128-bit AXI-Full bus, used for massive data transfers and capable of supporting high-throughput burst transactions with DDR memory. The 128-bit width was chosen to maximize the bandwidth, corresponding to the maximum width of the ZCU102 platform's HP ports.
- A set of GPIO signals with a tri-state model, designed to interact with external peripherals through the ATLAS virtualization layer. The signals follow the standard tri-state convention, where each logical GPIO consists of three signals: data input (I), data output (O), output enable (T), allowing bidirectional communication by leveraging the board's INOUT pins.

The subdivision between a lightweight control channel and a high-performance data channel is not accidental but reflects a consolidated design paradigm in digital systems. Traditionally, an accelerator exposes a register interface for configuration and monitoring (the control plane), and a separate data interface for processing (the data plane). The μ -VF architecture adopts and standardizes this model for each Lake. This guarantees an interface

that is not only efficient and familiar to hardware designers but is also the de-facto standard automatically generated by HLS tools.

The use of standard interfaces reinforces a "write-once, deploy-anywhere" development paradigm. Although bitstreams are not portable and require synthesis targeted at the specific board, the source design remains portable between different FPGAs that conform to the same interface specification. This architectural decoupling increases the reusability of tenant designs and facilitates integration into different target platforms.

While standardized bus interfaces like AXI are widely adopted industry norms on FPGA platforms, GPIO interfaces typically remain board-specific, with each platform defining its own pin assignments and I/O constraints. This creates a significant portability challenge that existing FPGA frameworks have not addressed [27][28].

The μ -VF approach addresses this gap: instead of constraining hardware logic to specific physical pins on the board, each Lake exposes virtual pins as logical placeholders. These virtual pins are dynamically mapped onto the physical GPIOs at runtime through ATLAS, our hardware-based virtualization layer.

This design offers several important advantages. First, it abstracts pin assignments at the board level, allowing developers to implement hardware without worrying about the specific physical I/O layout of the underlying platform. Consequently, user designs can be seamlessly migrated between different Lakes or even between heterogeneous FPGAs without requiring manual changes to the logic at the pin level. Furthermore, the ability to remap at runtime supports fault recovery by reassigning peripherals and enables dynamic resource allocation based on changing workload needs. Most critically, it provides secure and isolated access to peripherals in multi-tenant environments: a capability absent in existing FPGA virtualization frameworks.

ATLAS: Autonomous Tenant-Led Access to Signals.

The realization of these benefits requires a hardware mechanism capable of dynamically routing signals while maintaining electrical integrity and timing constraints. ATLAS implements this GPIO virtualization entirely in hardware within the programmable logic, preserving the

key property of embedded systems that I/O lines must be electrically and directly accessible from the user's hardware logic, with minimal latency and without software mediation.

As illustrated in Figure 2.2, ATLAS implements a crossbar-based routing architecture with integrated safety mechanisms:

- **Dynamic Routing Matrix:** At its core, ATLAS uses a configurable crossbar that maps virtual pins to physical PADs based on the 'owner_map' configuration register, which is dynamically updated at runtime by the software hypervisor. Each Lake exposes a fixed number of virtual pins, and ATLAS manages a total of N virtual pins (the sum of all Lakes). The owner_map uses values from 0 to N, where 0 indicates an unmapped pin and values from 1 to N map to virtual pins 0 to N-1, respectively. The routing logic is purely combinatorial in the output path: when a Lake drives a virtual pin, the signal propagates to the physical PAD through a combinatorial path, preserving the same electrical characteristics of a non-virtualized system.
- **Before establishing any connection,** ATLAS validates that each physical pin is assigned to at most one virtual pin. This validation is essential because multiple virtual pins cannot simultaneously drive the same physical PAD. The validation occurs in parallel on all pins: if multiple physical pins are configured to connect to the same virtual pin, only the first valid assignment is respected, while the others are safely disconnected.
- **Dual-Stage Output Control:** Output driving follows a two-level permission model. First, each Lake controls its own virtual pins through the standard tri-state signals, as already discussed. Second, the hypervisor maintains output enable flags for each individual physical pin. Consequently, a physical pin is driven only when both conditions are met. This dual control prevents unauthorized access to peripherals even if a malicious Lake tries to drive the outputs.
- **Anti-Glitch Pipeline:** The architecture intrinsically prevents glitches during reconfiguration through registered state updates. When the hypervisor updates the owner_map, the new configuration is first validated (uniqueness check), and then atomically applied on the next clock edge.

By maintaining purely combinatorial paths from virtual pins to physical PADs, ATLAS preserves the electrical and timing characteristics of direct physical connections. In practice, however, it is important to note that the signal passes through the crossbar logic (implemented via multiplexers), introducing a minimum propagation delay. This delay, although difficult to quantify exactly, is completely negligible for the vast majority of peripherals and communication protocols used in the embedded domain.

2.3.2 Shore and Isolation Mechanisms

As illustrated in Figure 2.3, the Shore implements the essential infrastructure for the secure sharing of resources and the isolation of the Lakes. The architecture employs a modular design where each Lake interfaces with dedicated components of the Shore to ensure both performance and security.

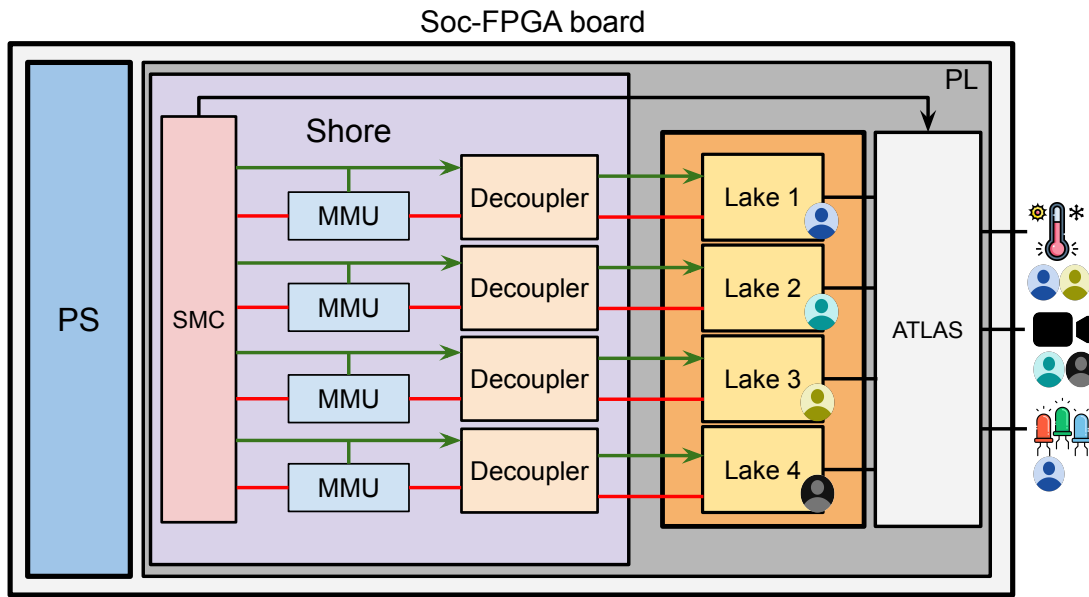


Figure 2.3: μ -VF Shore architecture. The Shore layer mediates between PS and PL, with per-Lake Memory Management Units (MMUs) that allow the hypervisor to configure memory access permissions for each tenant's hardware. Decouplers provide electrical isolation during reconfiguration. **Legend.** Red: AXI-Full bus; Green: AXI-Lite bus; Black: GPIO signals.

The Decoupler modules, positioned at the boundary between the Shore and each Lake, act as electrical isolation switches controlled by the PS hypervisor. When activated, a decoupler

completely disconnects the associated Lake from the Shore's infrastructure, preventing any signal from propagating in both directions. This isolation mechanism has a dual purpose: it guarantees system stability during partial reconfiguration, preventing unpredictable behavior while a Lake is being reprogrammed [2], and provides a safeguard against malicious or faulty designs that could try to interrupt system operation.

The Memory Management Units (MMU) manage memory access by user designs. The controllers are implemented directly in hardware to minimize performance overhead and latency, ensuring efficient communication with the DDR subsystem, and are at the same time carefully optimized to minimize resource usage within the fabric. In scenarios where a "role" operates as a bus master and initiates memory transactions, the controller imposes a strict isolation of accesses, allowing each "role" to access only the memory regions explicitly assigned to it by the hypervisor. These assignments are made dynamically: when a tenant requests the allocation of a buffer from their software container on the PS, the hypervisor allocates the memory accordingly and configures the hardware controller to grant access exclusively to the corresponding physical address range.

Smart Connectors (SMC) are used to connect the hardware components with the processing system, supporting both control operations and data transfers. Although this hardware provides the basic mechanisms for resource isolation and sharing, it relies on the software running on the PS to manage tenant allocation and coordinate the overall operation of the system.

2.4 Processing System Virtualization and Software Stack

Keeping in mind edge-class deployments, μ -VF adopts a completely on-device execution model in which each tenant's software stack is run directly on the same FPGA platform as their hardware logic.

This design choice is guided by fundamental constraints of edge environments:

First, running the control software on a remote host introduces unavoidable latency in hardware-software interactions. Consider a typical scenario where the control software must respond to a sensor event: the FPGA accelerator detects the event and must notify the remote

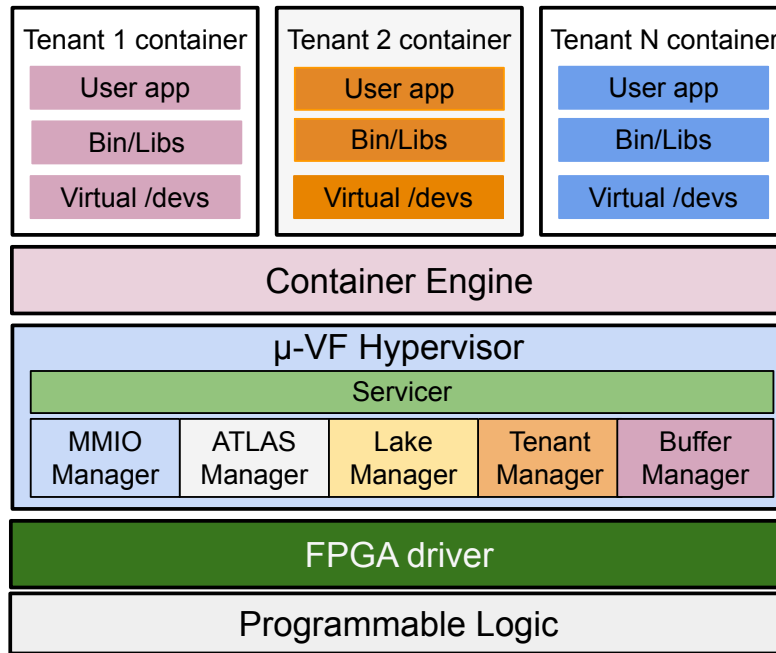


Figure 2.4: Software stack of μ -VF. Each tenant runs inside a containerized environment, supported by a container engine. The μ -VF hypervisor manages memory-mapped access, resource allocation, and partial reconfiguration through modular, multi-threaded services. A dedicated FPGA driver interfaces with the underlying programmable logic.

host through the network, the host processes the event and decides on an action, then sends the commands back to the FPGA. Even with low-latency networks, this round-trip communication adds milliseconds of delay. For edge applications that require response times on the scale of microseconds, such as motor control in robotics or real-time anomaly detection, these latencies are prohibitive.

Second, energy budgets at the edge are extremely limited. External host servers would add a significant power overhead (greater than that of the FPGA itself), making battery-powered deployments unfeasible.

As illustrated in Figure 2.4, the PS is partitioned into lightweight and isolated containers, each of which hosts a tenant's application code. This containerized architecture provides strong isolation between tenants while offering each of them a private and self-contained execution environment. Within this environment, the application code can interact closely and with minimal latency with the user's hardware logic in the fabric, leveraging shared memory buffers and fast communication paths that are only possible with on-device integration.

In addition to isolating tenants from each other, a containerized architecture also decouples each tenant from direct access to the underlying FPGA resources. This abstraction improves the overall security and robustness of the system, as tenants are allowed to interact with shared resources only through the supervision of the on-device hypervisor.

2.4.1 On-Device Hypervisor

The supervision layer is implemented as a modular service architecture, where specialized components manage different aspects of resource management and tenant coordination. The Servicer listens for incoming requests from tenant containers, maintaining dedicated communication channels for each tenant to support parallel and responsive interaction. Once a request is received, it routes the operation to the appropriate internal component. The Tenant Manager maintains the system state for each tenant, including authentication, authorization, and tracking of allocated resources. It enforces policy constraints, such as the maximum number of buffers, admissible hardware designs, accessible partial regions in the programmable logic (PL), the set of assignable GPIOs, and the physical slave interfaces available to the tenant. The application of these constraints is delegated to other specialized subsystems. The Lake Manager manages the secure deployment of the user's hardware designs in the assigned lakes. It coordinates with the decoupler modules during the transition phase to electrically isolate the target region, ensuring system stability and preventing transient faults during bitstream loading.

The Buffer Manager mediates access to shared DDR memory. It handles tenant memory allocation requests, configures the hardware MMUs to enforce access permissions, and ensures that user hardware designs can only read or write within the assigned address space.

The ATLAS Manager manages interactions with the I/O virtualization layer at the fabric level. It performs the dynamic mapping of each "role's" virtual GPIO pins to the physical pins of the board, based on system policy and runtime availability, allowing tenants to transparently access external peripherals without requiring static assignments or design modifications.

The MMIO Manager is responsible for managing the memory-mapped virtual devices exposed to tenant containers. Based on the "role" assigned to each tenant, the hypervisor dynamically sets access permissions and associates the appropriate virtual device nodes at

runtime. This mechanism ensures that tenants are granted access only to the memory regions corresponding to their assigned FPGA partition.

2.4.2 Container-Based Tenant Execution Flow

Figure 2.5 illustrates the software architecture and the interaction flow between tenant containers and hardware resources. The software stack implements a capability-based access control system that dynamically manages hardware permissions based on Lake assignments.

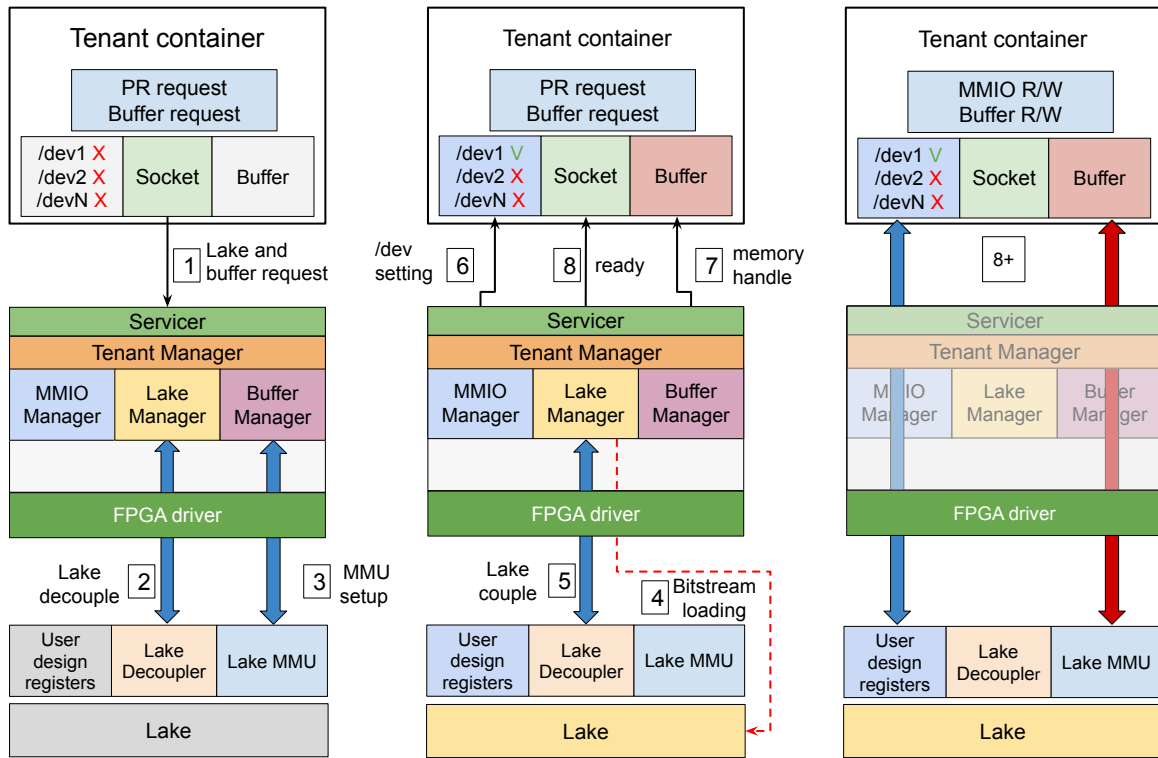


Figure 2.5: Software architecture and resource allocation flow for multi-tenant FPGA virtualization. The diagram shows the interaction between tenant containers, hypervisor services (MMIO Manager, Lake Manager, Buffer Manager), and FPGA hardware resources, with numbered steps (1-8) indicating the complete allocation sequence from resource request to direct hardware access. Blue arrows indicate AXI-Lite connections, dashed red lines represent ICAP/PCAP interfaces, and solid red lines show AXI Full connections.

To minimize runtime overhead and achieve near-native performance, μ -VF adopts a zero-copy communication strategy for all data-path operations between containers and hardware. This approach eliminates the latency and throughput penalties typically associated with

virtualization, by avoiding the copying of data through intermediate buffers or the mediation of the hypervisor.

For MMIO access, each container in the system is presented with N virtual MMIO devices (where N is the number of Lakes), which appear as /dev1 up to /devN in the container's filesystem. These devices provide direct memory-mapped access to the AXI-Lite control registers of the corresponding Lakes. When a container performs read/write operations on these device files, the CPU directly accesses the hardware registers via the kernel's memory mapping, with no data copying or hypervisor intervention occurring on the data-path. Access to these devices is strictly controlled via Linux file permissions (user/group ownership), ensuring that containers can interact only with their assigned Lake.

For high-throughput data transfers, μ -VF implements zero-copy shared memory buffers, supported by contiguous DDR regions. When a tenant requests memory allocation, the hypervisor allocates physically contiguous memory and configures the hardware MMU to grant the tenant's Lake exclusive access. The container receives a direct memory handle that allows both software and hardware components to access the same physical memory without any intermediate copying.

When a tenant's application needs to deploy an accelerator, it follows the sequence shown in Figure 2.5:

1. **Resource Request.** The tenant's application initiates a PR (Partial Reconfiguration) request via socket-based communication to the hypervisor, specifying: (i) the bitstream to load (previously registered with the hypervisor), (ii) the required memory buffers and their sizes, and (iii) any specific preferences on the Lakes.
2. **Lake Decoupling.** The hypervisor validates the request against security policies and resource availability. If approved, the Lake Manager decouples the target Lake to prepare it for reconfiguration.
3. **MMU Configuration.** The hypervisor configures the MMU with the physical addresses for the requested buffers via the Buffer Manager, setting up the memory mapping for the tenant's exclusive use.

4. **Partial Reconfiguration.** The FPGA driver initiates the partial reconfiguration, programming the Lake with the requested bitstream.
5. **Lake Re-coupling.** Once configuration is complete, the Lake is re-coupled to the system, making it ready for the tenant's use.
6. **Permission Grant.** The hypervisor updates the file permissions of the corresponding /dev device file, granting the requesting container exclusive access. For example, if Lake 1 is assigned, the container gains read/write access to /dev1 through changes to Linux user/group ownership.
7. **Buffer Handle Transfer.** The hypervisor returns the handle of the contiguous memory buffer created for the tenant.
8. **Ready Resource Notification.** The hypervisor notifies the client that the resources are ready for use.

This design provides each tenant with the illusion of exclusively owning the FPGA, while the hypervisor maintains system-level coordination and security. The socket-based control plane ensures secure resource negotiation, while the permission-based data plane enables high-performance hardware access without hypervisor intervention in the critical path.

CHAPTER

3

Implementation Details

This chapter presents the implementation details of the μ -VF framework. Rather than providing a comprehensive account of every implementation aspect, the focus is placed on the elements that are particularly novel, challenging, or critical to the system's functionality. For readers interested in a complete view of the codebase, the full implementation is publicly available on GitHub [36].

3.1 Experimental Platform and Tools

The implementation and validation of the μ -VF framework were conducted on two SoC-FPGA based hardware platforms: the AMD Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit, used for the main development and performance evaluation, and the AMD Zynq-7000 ZedBoard, used for portability tests.

The development of the hardware stack was carried out using the AMD Vivado Design Suite v2024.2, running on an Ubuntu 24.04 system.

For the on-device software environment, both platforms run the PYNQ v2.7.0 distribution, which provides the basic operating system and interaction libraries. The software-level isolation of tenants was implemented using Docker containerization technology.

The hypervisor and APIs of μ -VF, in this prototype phase, were developed in the Python language.

3.2 Hardware Stack Implementation

In this section, the most important implementation details of μ -VF are illustrated. The steps described in the following sections apply equally to the platforms used in this thesis work.

3.2.1 Basic Processing System Configuration

The first step in the implementation of the hardware design consists of configuring the Processing System (PS). After instantiating the Zynq UltraScale+ PS IP core in Vivado's IP Integrator, the basic configuration is automatically applied via the Run Block Automation function, which adapts the parameters to the specificities of the ZCU102 board.

Subsequently, this basic configuration was modified to satisfy the specific requirements of μ -VF. In particular, six AXI High-Performance (HP) ports were enabled, two masters and four slaves, in order to dedicate a high-speed data channel to each of the four Lakes. A GPIO channel was also enabled on the PS, necessary to allow the hypervisor to drive the control signals of the Decoupler modules. Finally, the clock frequency for the Programmable Logic was set to 100 MHz for the entire framework and for all experimental evaluations.

3.2.2 Creation of Lakes

As defined in the previous chapter, the Lakes are the reconfigurable partitions destined to host the hardware designs of the tenants. At the implementation level, the challenge consists of creating regions on the fabric that are mutually isolated and dynamically reconfigurable, without interfering with the static infrastructure or with the other tenants.

This result was obtained by leveraging the paradigm of Dynamic Partial Reconfiguration (DPR), known in the AMD/Xilinx ecosystem as Dynamic Function eXchange (DFX).

Although DFX technology can be applied to different design flows, for this thesis work the approach based on IP Integrator was chosen. Since the system was assembled using block design, the DFX was enabled using the Block Design Container construct, an advanced feature of Vivado's IP Integrator. A BDC extends the concept of a hierarchical block, allowing a portion of the design (composed of one or more IPs) to be encapsulated in a container

that in turn acts as a block design in its own right. The fundamental characteristic of a BDC is that it can be designated as reconfigurable, allowing the designer to create different alternative versions of the logic contained inside it. By enabling the DFX on a BDC, it is therefore possible, at runtime, to dynamically replace the implementation of the block, loading a different partial bitstream to change its behavior without altering the rest of the system.

The generation of a block design container occurs by designating an IP block as such (BDC). As discussed previously, a fundamental requirement of Partial Reconfiguration is the immutability of the interface between the static and dynamic region. To guarantee this constraint in a programmatic way, a placeholder IP was created that acts as a "mold" for the interface of each Lake.

The first phase had the objective of generating the complex AXI bus interfaces and for this purpose a simple module was created in Vitis HLS. However, HLS compilers are designed to aggressively optimize the design, eliminating any logic or interface that does not contribute to the final result. To prevent the AXI interfaces from being removed during optimization, it was necessary to implement a minimal functional body that actively used them. For this purpose, a simple adder was inserted, whose only function is to create a dependency on the control and data channels, thus forcing the tool to generate them correctly.

To explicitly define the nature and structure of these interfaces, pragma HLS directives were used in the following source code:

Listing 3.1: Extract of the HLS placeholder code

```
1 typedef ap_uint<BUS_WIDTH> uint128_t;
2 void vector_scalar_sum(
3     uint128_t *input_vector,
4     uint128_t *output_vector,
5     int scalar,
6     int size
7 )
8 {
9     #pragma HLS INTERFACE m_axi port=input_vector offset=slave
```

```
    bundle=gmem0
10 #pragma HLS INTERFACE m_axi port=output_vector offset=slave
    bundle=gmem1
11 #pragma HLS INTERFACE s_axilite port=scalar
12 #pragma HLS INTERFACE s_axilite port=size
13 #pragma HLS INTERFACE s_axilite port=return
14
15 ...
16 }
```

To instruct Vitis HLS to generate the required 128-bit AXI-Full interface, a custom data type (`uint128_t`) was first defined to set the bus width. This type was then used for the `input_vector` and `output_vector` arguments, declared as pointers to indicate to the tool that these are interfaces to memory.

The `pragma HLS INTERFACE m_axi` directive is the key command that instructs the compiler to generate a master AXI4 interface (AXI-Full). The `bundle` keyword is crucial: by assigning different bundle names (`gmem0` and `gmem1`), the creation of two physically separate AXI-Full ports is forced, one for input and one for output, thus realizing the distinct data channels provided by the architecture.

For the control plane, the scalar arguments `scalar` and `size`, along with the function's control signals (`return`), were associated with an AXI-Lite interface via the `pragma HLS INTERFACE s_axilite` directive. Since a different bundle was not specified for these signals, Vitis HLS automatically groups them into a single 32-bit AXI-Lite interface, which acts as a control register for the entire accelerator.

At the end of the Vitis HLS synthesis and packaging process, the module is exported as a reusable IP core. This IP can then be imported into the Vivado catalog and instantiated within a block design via the IP Integrator. The resulting block, as shown in Figure 3.1, faithfully exposes the AXI-Full and AXI-Lite interfaces defined previously via the HLS pragmas.

To mark the block as a block design container, it is necessary to first designate it as a

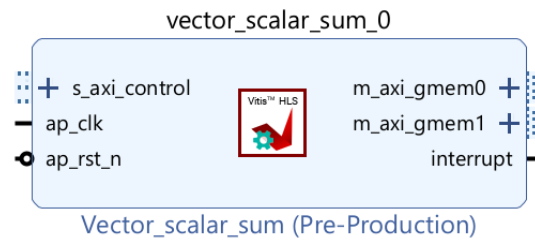


Figure 3.1: Placeholder IP generated through Vitis.

Hierarchical Block. During this phase, it is possible to name the block, for clarity, a systematic nomenclature was adopted, such as PR_N_bit_0, to identify the partition (N) and the specific implementation (bit.0 for the placeholder). This operation can be carried out via GUI or using TCL commands, as follows.

Listing 3.2: Hierarchical Block definition using TCL

```

1 current_bd_design [get_bd_designs design_1]
2 create_bd_cell -type ip -vlnv xilinx.com:hls:vector_scalar_sum:1.0
   vector_scalar_sum_0
3 endgroup
4 group_bd_cells PR_0_bit_0 [get_bd_cells vector_scalar_sum_0]

```

To mark the module as a block design container, it is necessary to first validate the design through Vivado. For this purpose, it is strictly necessary to appropriately connect the clock pin of the module. Subsequently, via GUI or using the TCL script reported below, it is possible to designate the module as a BDC.

Listing 3.3: BDC setup using TCL

```

1 set curdesign [current_bd_design]
2 create_bd_design -cell [get_bd_cells /PR_0_bit_0] PR_0_bit_0
3 current_bd_design $curdesign
4 set new_cell [create_bd_cell -type container -reference
5 update_compile_order -fileset sources_1

```

The final step in the creation of the block design placeholder consists in adding the virtual I/O pins as an interface. Since Vitis HLS is not optimal for defining single-signal ports, the addition can be done by accessing the block design container and manually instantiating, for example via GUI, the GPIO pins with the standard adopted by the architecture.

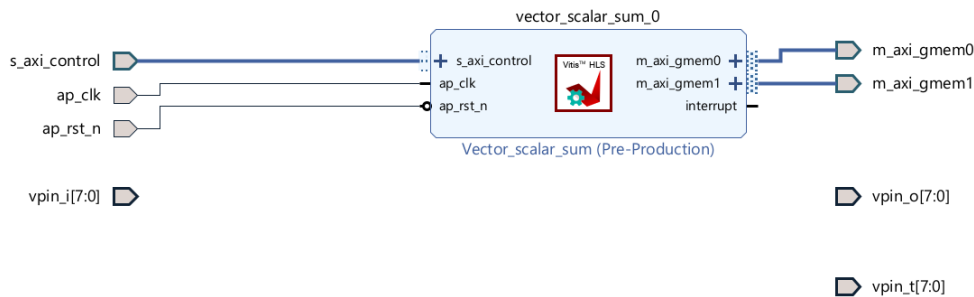


Figure 3.2: GPIO interfaces definition inside a BDC.

Once the placeholder IP is finalized, which acts as a "mold" for the interface of each partition, the Block Design Container that contains it must be officially designated as reconfigurable. This is done by enabling the Dynamic Function eXchange (DFX) option on the container itself, an operation that can be performed both via the Vivado graphical interface and programmatically with the following Tcl command:

```

1 set_property CONFIG.ENABLE_DFX 1 [get_bd_cells /pr_block_n]

```

At this point, the container represents in all respects a reconfigurable region ready to host the designs of the tenants. The last step of the development flow for the user consists in inserting their custom logic inside the container, replacing the placeholder module. The modalities of this replacement will be illustrated in the following sections.

It is important to note that the entire procedure described here for a single Lake can be automated using Tcl scripts, so as to quickly and reproducibly create all the other reconfigurable partitions of the system.

3.2.3 Isolation Mechanisms via Decouplers and MMUs

As discussed previously, to ensure operational security in a multi-tenant environment, μ -VF implements rigorous isolation mechanisms at both the software and hardware levels. Hardware isolation, in particular, is realized through two key components integrated into the Shore: the Decouplers and the MMUs.

The Decouplers have the task of electrically isolating a Lake from the rest of the infrastructure, operating as switches controlled by the hypervisor. This isolation is fundamental in two scenarios. The first is during partial reconfiguration: a Lake in the programming phase could emit spurious signals on the buses, causing instability to the entire system; the Decoupler prevents this risk by physically disconnecting the partition. The second is a security mechanism: the hypervisor can decide to isolate a Lake in case of anomalous or malicious behavior, such as an attempt to access unauthorized memory areas, thus protecting the integrity of the system.

In the architecture of μ -VF, the Xilinx Decoupler IP [37] was used, which can be instantiated in the block design via the IP integrator or the following tcl commands.

Listing 3.4: PL-MMU integration in the block design

```
1 startgroup
2 create_bd_cell -type ip -vlnv xilinx.com:ip:dfx_decoupler:1.0
   dfx_decoupler_4
3 endgroup
```

Although the IP offers the possibility of being controlled via an AXI interface, a lower overhead solution was opted for. In the Decoupler's configuration, the control option via a direct enable signal, driven by a GPIO pin of the Processing System, was enabled. This implementation choice, although it requires dedicated routing, allows to avoid the instantiation of further bus interconnection logic (such as Smart Connectors), contributing to minimizing the area occupied by the static infrastructure on the PL.

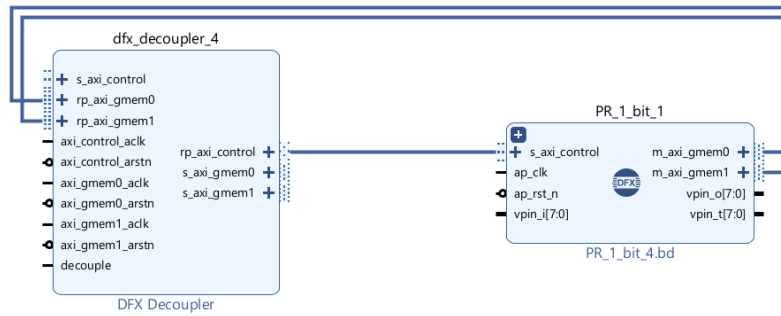


Figure 3.3: BDC (reconfigurable module) interfaced with a Decoupler.

Figure 3.3 illustrates the instance of a Decoupler with the necessary interfaces. Through the decoupler, the Lake can be interfaced via the three bus channels to the processing system, however, while the control bus (AXI-Lite) can be directly connected to the Processing System, allowing the hypervisor to validate the requests, the data channels (AXI-Full) represent a security challenge.

Requests for access to DDR memory originating from an accelerator inside a Lake (master transactions) would escape direct software control. A malicious or faulty tenant could therefore try to access unauthorized memory areas.

To solve this vulnerability, hardware Memory Management Units (MMU) were used, one for each Lake. Each MMU, interposed on the data path, acts as a memory firewall. The hypervisor, through a dedicated AXI-Lite control interface, can dynamically configure the access rules for each MMU, specifying the address ranges that the corresponding Lake is authorized to access, thus guaranteeing a rigorous and secure isolation. More precisely, the allocation process is entirely mediated by the software: a tenant, via API, requests one or more contiguous memory buffers from the hypervisor's Buffer Manager. The hypervisor

verifies the request against the tenant's policies, allocates the requested physical memory regions in the DDR and, finally, programs the hardware MMU of the corresponding Lake. This configuration, performed via an AXI-Lite interface, sets the access rules, specifying the exact address ranges to which the accelerator is authorized to access and blocking any attempt to access outside of these limits. The current implementation of the MMU allows to configure up to 15 distinct memory regions for each Lake.

3.2.4 ATLAS

The ATLAS module provides a flexible mapping between physical pins and virtual pins, a fundamental feature of the μ -VF framework. Each physical pin is assigned to a virtual pin according to a configurable mapping vector (`owner_map`). The module ensures that each virtual pin is driven by at most one physical pin at a time, preventing bus contention and guaranteeing deterministic behavior. Output driving is managed by combining the physical pin output enables (`pad_oe`) and the virtual pin tristate signals (`vpin_t`). This approach allows multiple tenants to safely share the same physical pins. Physical pin inputs can optionally be synchronized to the system clock, depending on the `SYNC_INPUT` parameter, providing flexibility for different timing requirements. The core functionality can be summarized in the following annotated pseudocode:

Algorithm 1 Simplified pseudocode of the `crossbar_map_tri` module

```

1: for each physical pin  $i$  do
2:    $vpin\_idx \leftarrow owner\_map[i]$ 
3:   if  $vpin\_idx$  is valid and unique then
4:      $pad\_drive \leftarrow pad\_oe[i]$  AND NOT  $vpin\_t[vpin\_idx]$ 
5:      $pad\_io[i] \leftarrow vpin\_o[vpin\_idx]$  if  $pad\_drive$  else 'Z'
6:   end if
7: end for
8: if SYNC_INPUT then
9:   on rising_edge(clk):  $pad\_in\_reg \leftarrow pad\_io$ 
10: else
11:    $pad\_in\_reg \leftarrow pad\_io$ 
12: end if
13: for each virtual pin  $j$  do
14:    $vpin\_i[j] \leftarrow$  OR of all  $pad\_in\_reg[i]$  mapped to  $j$ 
15: end for

```

The annotated pseudocode highlights the main operations and control flow, without burdening the reader with full VHDL details. The complete implementation is publicly available on GitHub.

3.2.5 Integration of the Hardware Stack

The interconnection infrastructure of the hardware components deserves a detailed analysis, as it is responsible for both the performance of the system and a significant part of the occupied area, as will be seen in the evaluation chapter. To realize this infrastructure in μ -VF, the AXI SmartConnect IP core was used.

This IP, which allows connecting one or more AXI memory-mapped master devices to one or more memory-mapped slave devices, operates in two main modes: a low-area mode, which optimizes resources, and a high-performance mode, which prioritizes throughput. The architecture of μ -VF strategically leverages both: the low-area mode is used for the interconnection of the AXI-Lite control buses, where bandwidth is not critical, while the high-performance mode is used for the AXI-Full data channels, to maximize throughput with memory.

It is crucial to note that the selection of these modes is not an explicit configuration

option. Vivado automatically infers the low-area mode only in topologies with 32-bit AXI-Lite buses. Any other configuration activates the high-performance mode by default. Therefore, although Vivado's automatic connection tool uses the SmartConnects, to obtain the hybrid and optimized architecture of μ -VF, a manual instantiation and an accurate connection topology were necessary, in order to force the tool to generate the most efficient interconnection for each communication plane.

More precisely, the processing system equipped by the ZCU102 offers two master HP ports, while in a configuration for 4 tenants the hardware stack of μ -VF provides for the connection of 9 slave devices to the processing system, specifically the 4 decouplers connected in turn to the slaves, 4 MMUs (one for each tenant) and ATLAS. However, while the protocol used by the decouplers and ATLAS is AXI-Lite, the MMUs use AXI-Full.

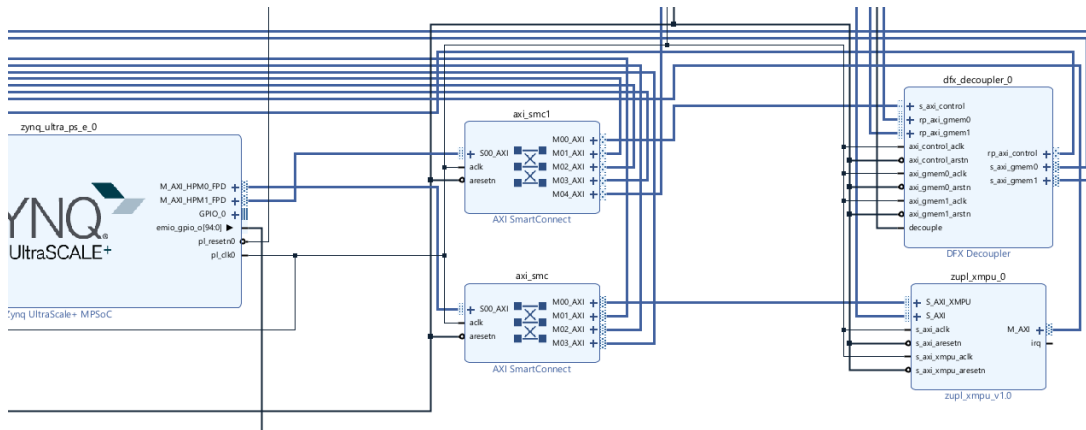


Figure 3.4: The two main SmartConnects used to interconnect the AXI slaves. Separating AXI-Lite and AXI-Full slaves into distinct SmartConnects enables μ -VF to reduce area utilization.

To solve this misalignment and at the same time optimize resources, two distinct AXI SmartConnect IPs were used, one for each master port of the PS. The slaves were partitioned strategically based on the AXI protocol used, as illustrated in Figure 3.4:

- A first SmartConnect connects a master port of the PS to all slaves that use the AXI-Lite protocol (the Decouplers and ATLAS). Since this topology is homogeneous, Vivado automatically infers and implements the low-area mode, minimizing resource consumption for the control plane.
- A second SmartConnect connects the remaining master port of the PS to the four MMUs,

which require the AXI-Full protocol, operating consequently in the high-performance mode.

For the data plane, the Processing System of the ZCU102 exposes four AXI High-Performance (HP) slave ports, allowing to dedicate a high-throughput connection to each of the four tenants. Although the topology is conceptually 1-to-1 between each Lake and an HP port, it was necessary to interpose an AXI SmartConnect for two implementation reasons.

First, the SmartConnect acts as a fusion unit: to simplify the design for the tenant, the Lake interface exposes logically separate AXI4 read and write channels, which are then combined by the SmartConnect into a single physical connection to the PS port.

Second, and more critically, the SmartConnect acts as a protocol adapter. The MMU IP, interposed between the Lake and the PS, is designed to manage multi-ID AXI transactions, thus exposing the AWID and ARID signals. However, in the architecture of μ -VF, with a dedicated MMU for each tenant, these identification signals are redundant and are not used by either the Lake interface or the PS interface. The SmartConnect automatically manages this misalignment, adapting the protocol and guaranteeing compatibility between the components.

Finally, for the control of the Decouplers, a minimum overhead solution was adopted. Since their management requires only simple enable/disable signals, it was decided to drive them directly via GPIO pins of the Processing System. This decision avoided the need to instantiate a dedicated AXI bus interface, contributing to reducing the overall area occupied by the static infrastructure.

3.2.6 Generation of the Abstract Shell and Floorplanning

Once the static infrastructure is completed and all placeholders for the Lakes are arranged, it is possible to instruct Vivado in the generation of an Abstract Shell and subsequently manually define the floorplanning.

As previously discussed, the standard Partial Reconfiguration (DFX) flow allows generating new partial bitstreams for reconfigurable modules without ever having to re-implement the static part of the design, which is locked after the first implementation. However, this standard process requires that, for the compilation of each new module, the entire and heavy database

of the static design is loaded as a "context" of reference.

To optimize this flow, modern EDA suites introduce the concept of an Abstract Shell. An Abstract Shell is a minimal and lightweight representation of the static design. It does not contain the entire implementation of the static part, but only the information strictly necessary to compile a new reconfigurable module, such as the physical location and temporal constraints of the boundary interfaces. Loading a lightweight shell instead of the complete design accelerates the compilation, reducing its times.

Having created an HDL wrapper for the general block design and generated the output products of all block designs (including the block design containers), it is possible to start the Vivado wizard for Dynamic Function eXchange. During the wizard, the tool provides an overview of all reconfigurable modules and their respective Lakes and allows programming standard configurations, useful for generating complete bitstreams with pre-assigned instances of reconfigurable modules. If in the first instance you only want to generate the general bitstream with the placeholders, postponing the generation of the partial bitstreams of the user modules to later moments, a single basic configuration can be created by predefining the placeholders as standard modules, as visible in the figure.

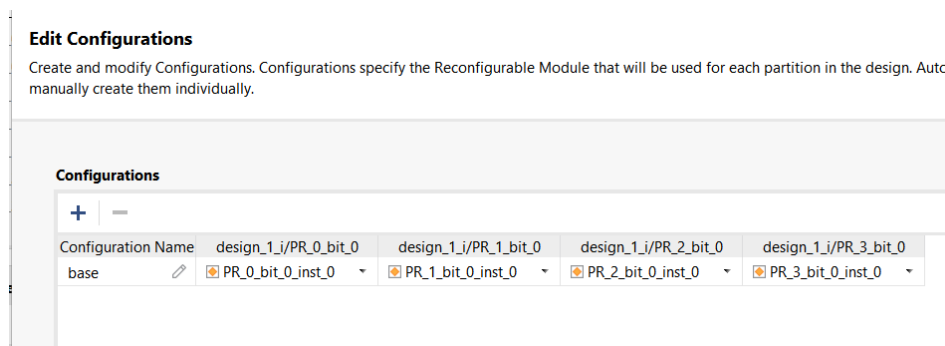


Figure 3.5: Implementation configuration setup through Vivado DFX Wizard

It is of fundamental importance to define the use of the Abstract Shell paradigm in the creation of a new configuration run. This way, Vivado can be instructed to create an implementation, which will act as the parent implementation for the subsequent ones, i.e., for the reconfigurable modules.

Once the wizard is finished, the synthesis process can be started. The complete implemen-

tation (Place Route) is deliberately postponed, as it is necessary to intervene in this phase to manually specify the floorplanning.

Once the synthesis process is finished, it is possible to access the device view to proceed with manual floorplanning. This phase consists of defining placement constraints to specifically place the Lakes (and consequently, by exclusion, the static regions) on the FPGA fabric. As visible in Figure 3.6, the useful area of the ZCU102 has a shape with geometric irregularities. If an approach were adopted, as in FOS, aimed at creating partitions with an exactly identical number of resources, this irregularity would drastically limit the total area allocable to users. To overcome this limitation, in μ -VF the partitions are not perfectly equal, but reasonably similar, as illustrated in the table 4.2 of the Evaluation Chapter. This approximation allows μ -VF to assign about 30% more resources to users than previous work (FOS [27]), allocating a total of 85% of the fabric.

The mechanism used in Vivado to assign a portion of an FPGA to a module is the pblock, a construct that defines a set of physical resources (such as LUTs, BRAMs, etc.) within one or more rectangular areas. A pblock is associated with a hierarchical cell of the design to constrain its placement.

By way of example, the creation process can be performed interactively via the Vivado GUI: starting from the Netlist view, the designer selects the hierarchical cell of interest (in this case, the Block Design Container) and draws a rectangular area directly on the device view. The tool then associates the physical resources contained in that area with the newly created pblock.

Once the pblock for a reconfigurable partition is created, it is strictly necessary to modify its properties. The key property to set is HD.RECONFIGURABLE on the associated hierarchical cell. Setting this property to TRUE instructs Vivado to automatically apply a series of fundamental constraints for the DFX flow, including:

- DONT_TOUCH: Prevents logical optimizations across the partition boundaries.
- EXCLUDE_PLACEMENT: Prevents static logic from being placed inside the reconfigurable area.

- **CONTAIN_ROUTING**: Forces all the module's routing to remain confined within the pblock's borders, ensuring isolation.

To maximize the area for users, the floorplanning of μ -VF configured for 4 tenants on the ZCU102 is the one illustrated in the Figure 3.6. As can be seen, in the floorplanning technique used, space was left for the static part (light blue cells) of the design in the immediate vicinity of the processing system, since the Shore is in close contact with it. Between one Lake and another, represented by the green cells, a minimal area was left for interconnections with Shore and ATLAS.

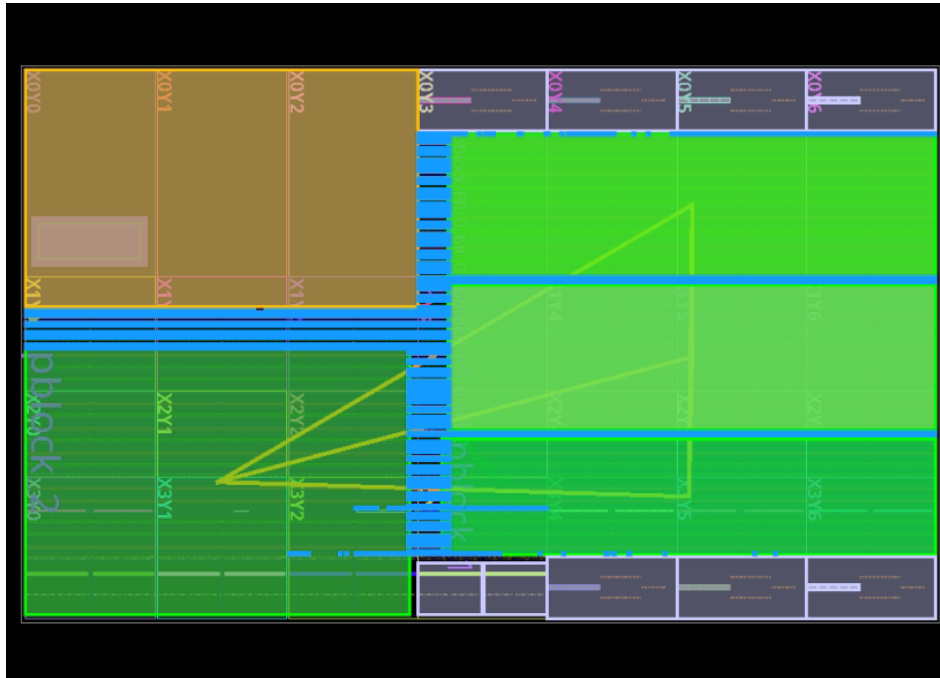


Figure 3.6: Floorplanning view after synthesis and implementation. Lakes are represented by green areas, the processing system occupies the orange area, while the hardware stack (Shore and ATLAS) is shown in blue.

Once the pblocks are defined, it is sufficient to save the constraints defined with the pblocks and proceed with the place route and the generation of the complete bitstream.

3.2.7 Adding user hardware design to the Lakes

Once the Block Design Containers that act as placeholders for the Lakes are defined, the next step consists in integrating the specific hardware designs of the tenants. Since the objective

of this thesis is the definition of an enabling architecture rather than a complete end-to-end system, this section illustrates the manual process of integration. It should be noted, however, that the entire flow can be automated via Tcl scripts, an aspect that falls within the future developments of the project. The process of integrating a new user design into a Lake is divided into two main phases.

The first phase consists of creating a new reconfigurable variant (or Reconfigurable Module) for the Block Design Container that represents the Lake. This operation, which can be performed both graphically and via Tcl scripts, generates a new empty block design (e.g., PR_0_bit_n for Lake 0) that automatically inherits the immutable interfaces defined by the container. This new design is then added to the list of possible implementations for that reconfigurable partition; in this example, PR_0_bit_n is added in addition to the placeholder.

Listing 3.5: New Reconfigurable Module generation through TCL

```

1 current_bd_design [get_bd_designs design_1]
2 set curdesign [current_bd_design]
3 create_bd_design -boundary_from_container [get_bd_cells /PR_0_bit_0]
   PR_0_bit_n
4 set_property -dict [list CONFIG.LIST_SYNTH_BD
   {PR_0_bit_0.bd:PR_0_bit_n.bd} CONFIG.LIST_SIM_BD
   {PR_0_bit_0.bd:PR_0_bit_n.bd}] [get_bd_cells /PR_0_bit_0]
5 current_bd_design $curdesign
6 current_bd_design [get_bd_designs PR_0_bit_n]
7 update_compile_order -fileset sources_1

```

In the second phase, this new variant is populated with the user's logic. Inside the block design of the variant, the custom IP is instantiated (for example, a subtractor, as shown in Figure 3.7) and its ports are connected to the pre-existing AXI and GPIO interfaces.

Once the design inside the Block Design Container has been validated, it is possible to proceed with the generation of the partial bitstream for the new module. This process assumes that the floorplanning and the initialization of the DFX flow have already been

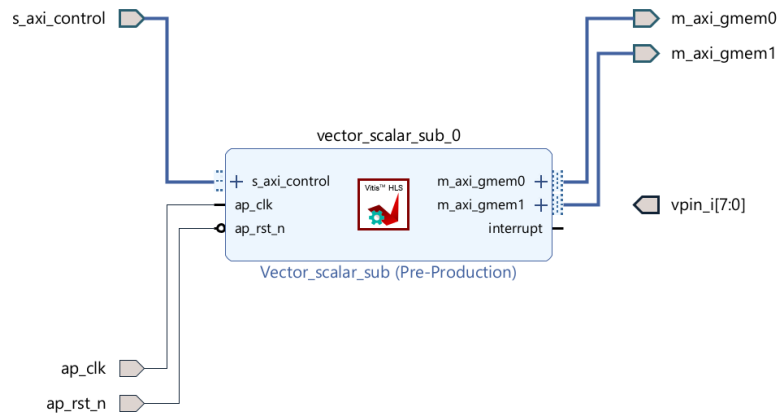


Figure 3.7: User hardware design, i.e., a subtractor, wrapped in the BDC.

completed for the main design, as described in the subsection 3.2.6. Having already a complete implementation of the system (the "parent run"), the generation of a partial bitstream for a new reconfigurable variant occurs by creating a so-called "child implementation". Through the Dynamic Function eXchange Wizard of Vivado, it is possible to define a new configuration run specifically for the new module.

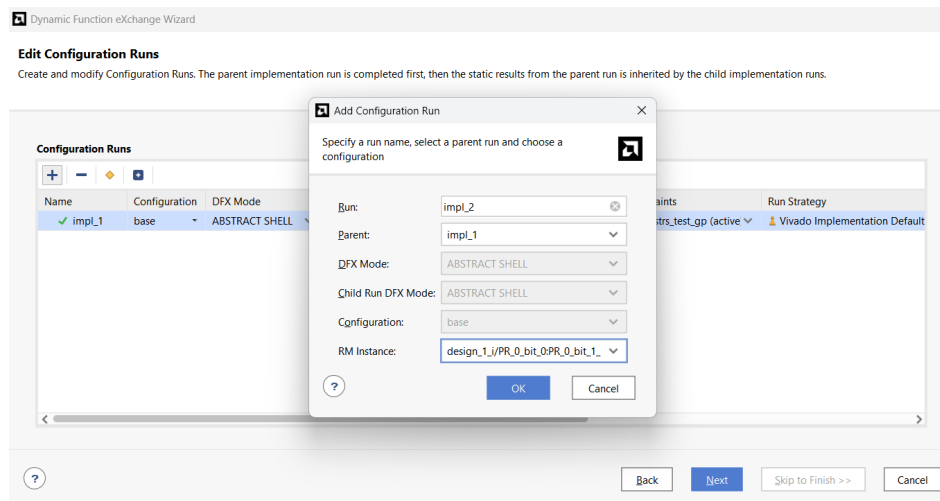


Figure 3.8
Child implementation run definition through DFX Wizard

As shown in Figure 3.8, it is fundamental to associate this new execution with the parent implementation (impl_1 in the example) and select the variant of the reconfigurable module for which you want to generate the partial bitstream.

This operation, which at a programmatic level translates into a Tcl command like the following, instructs Vivado to implement solely the logic of the new module.

```
create_run impl_2 -parent_run impl_1 -flow {Vivado Implementation  
2025} -rm_instance design_1_i/PR_0_bit_0:PR_0_bit_1_inst_0
```

Once created, the implementation can be started via the graphical interface or with the command `launch_runs impl_2 -jobs 24`. The fundamental advantage of this approach is that the process is not executed on the entire design, but only concerns the logic of the new reconfigurable module. The static parts and the other variants are not re-implemented, with a consequent and drastic saving of compilation time and computing resources.

3.3 Software Stack Implementation

The μ -VF software stack is composed of a hypervisor for general resource management, client-side API modules for resource access, and kernel modules that manage virtual devices to enable zero-copy access to MMIO and buffers.

3.3.1 Container startup and Hypervisor configuration file

The hypervisor's behavior is defined in a YAML configuration file that specifies the credentials of admitted tenants, resource limits per tenant, and system parameters, as well as the properties of the PR zones.

In the properties of the PR zones, an ID, the MMIO address range, and the reference decoupler GPIO pin are specified for each one. System parameters define the number of PR zones and the locations of the user and static bitstreams.

Listing 3.6: Excerpt of the hypervisor configuration file

```
1 pr_zones:
2   - zone_id: 0
3     gpio_pin: 0
4     address_ranges:
5       - [0xB0000000, 0x10000]
6   - zone_id: 1
7     gpio_pin: 1
8     address_ranges:
9       - [0xB0010000, 0x10000]
10 # ...
```

Subsequently, the tenant properties are extended with an ID, API key, UID, as well as their permitted resource quotas (amount of allocatable memory, how many bitstreams they can allocate, and which ones).

Listing 3.7: Excerpt of the hypervisor configuration file

```
1 tenants:
2   - id: tenant1
3     uid: 1001
4     api_key: test_key_1
5     max_overlays: 2
6     max_memory_mb: 512
7     allowed_bitstreams:
8       - PR_0_sum.bit
9       ...
10    allowed_pr_zones: [0, 1, 2, 3]
```

Each container is then launched with parameters matching its configuration entry and mounts its dedicated char device. For example, `tenant1` mounts `/dev/pynq_mem.tenant1`,

tenant2 mounts `/dev/pynq_mem_tenant2`, both to the same internal path `/dev/pynq_mem`. This provides API uniformity while maintaining complete buffer isolation at the kernel level.

Listing 3.8: Startup of a tenant Docker container

```
1 docker run -it --rm \  
2   --user 1001:1001 \  
3   --device /dev/pynq_mem_tenant1:/dev/pynq_mem \  
4   ...
```

All containers mount all four UIO devices (`/dev/uio10-13`), one per PR zone. However, initial ownership is root, making them inaccessible. When tenant1 loads an overlay and receives PR zone 0, the hypervisor executes `chown /dev/uio10 1001:1001`, granting exclusive access. Tenant2 (UID=1002) still cannot open `/dev/uio10` due to permission mismatch. Socket directory (read-only): `/var/run/pynq` contains Unix sockets for all tenants. The read-only mount prevents containers from creating new sockets, while individual socket permissions (tenant1.sock owned by UID=1001) ensure tenant1 can only connect to its own endpoint.

3.3.2 APIs and Hypervisor overlay resources assignment

The goal of the software stack, beyond targeted resource management, is ease of use for the end-user. Indeed, users should have the abstraction of using the FPGA in a conventional manner. With reference to a PYNQ-based system, the objective of the hypervisor and client APIs is to provide the user with the same user experience they would have when using the classic PYNQ APIs. For this reason, each container in the architecture is provided with a set of modules (overlay, mmio, allocate, register_map), similar to how it is done in PYNQ. The object-oriented modules present methods with interfaces identical to those of PYNQ; however, the implementation is completely different to ensure correct interaction with the hypervisor running on the host, which is clearly outside the containers.

Take the simple adder IP as an example. A user on PYNQ would use a similar code snippet for execution. In this code snippet, they would request to load the overlay (which, in PYNQ's terminology, is the bitstream), create an MMIO object to access their IP's slave registers, and

allocate a CMA buffer to allow the IP to access data in DDR. In μ -VF, the process is perfectly identical, with the hypervisor having to efficiently manage these instructions while considering a secure, multi-tenant environment.

```
1 ol = Overlay("PR_0_sum.bit")
2 input_buf = allocate(shape=(104,), dtype=np.int32)
3 output_buf = allocate(shape=(104,), dtype=np.int32)
4 input_buf[:] = np.arange(104)
5 mmio = MMIO(0x000000)
6 mmio.write(0x10, input_buf.physical_address & 0xFFFFFFFF)
7 mmio.write(0x1C, output_buf.physical_address & 0xFFFFFFFF)
8 mmio.write(0x28, 100)
9 mmio.write(0x30, 104)
10 mmio.write(0x00, 0x01) # START
11 print(f"Output: {output_buf[:10]}")
```

More precisely, assignment instructions like `allocate` and `overlay` must be handled by the hypervisor, whereas non-assignment operations occur through virtual devices to avoid gRPC overhead. Indeed, while gRPC introduces an acceptable overhead during the resource assignment phase, this overhead is too high during the active phase. It would render the efforts to optimize the Shore for best performance useless by introducing latency and lowering throughput due to RPC serialization.

As can be seen from the GitHub repository related to this thesis work, the management part of the hypervisor was developed in Python following the object-oriented programming paradigm. Developing an essential component like the hypervisor in Python might seem strange compared to a counterpart developed in C. It should be noted, however, that the hypervisor comes into play during the resource assignment phase: handling requests to load a bitstream, requests for buffer allocation, and the setup of containers and virtual devices. Therefore, it does not perform actions on latency-sensitive tasks, which are instead handled by low-level kernel modules.

Starting from the example in the listing, the corresponding actions by the client APIs and

the Hypervisor are illustrated below. The code in the listing is executed within a container, which has already been started and has the virtual devices for CMA and MMIO access created by the hypervisor before the container's startup. The container has a previously set UID/GUID, by which it is recognized and upon which the permissions for accessing the virtual devices are set.

Before any API operation, the container must authenticate with the hypervisor. This occurs automatically on first API usage.

```
1 from pynq_proxy import Overlay
2
3 # Connection.__init__() reads environment
4 self.tenant_id = os.environ['TENANT_ID'] # 'tenant1'
5 api_key = os.environ['PYNQ_API_KEY']    # 'test_key_1'
6
7 # On first API call, connect() is invoked
8 def connect(self):
9     socket_path = f"/var/run/pynq/{self.tenant_id}.sock"
10    self.channel = grpc.insecure_channel(f'unix://{socket_path}')
11    self.stub = PYNQServiceStub(self.channel)
12
13    # Authenticate
14    request = pb2.AuthRequest(tenant_id='tenant1',
15                               api_key='test_key_1')
15    response = self.stub.Authenticate(request)
16    self.token = response.session_token
```

The server validates credentials by comparing the provided API key against the value stored in config.yaml. Upon successful validation, it generates a unique session token stored in the active_tokens dictionary, mapping token to (tenant_id, expiry_timestamp). The token enables fast validation in subsequent calls: the server extracts it from gRPC metadata and performs a simple dictionary lookup rather than re-validating credentials.

```
1     # In TenantManager.authenticate()
2 def authenticate(self, tenant_id='tenant1', api_key='test_key_1'):
3     # Verify tenant exists
4     if tenant_id not in self.config:
5         return None
6
7     # Compare API key with configuration
8     config_key = self.config['tenant1'].api_key # From config.yaml
9     if api_key != config_key:
10        return None
11
12    # Generate session token
13    token = f"token_{uuid.uuid4().hex}"
14    expiry = time.time() + 3600 # 1 hour validity
15
16    # Store for future validation
17    self._active_tokens[token] = (tenant_id, expiry)
18
19    return token
```

When the user requests to load a bitstream, the client-side APIs prepare a gRPC request which is then transmitted via socket to the hypervisor, with an authentication token added. The request is handled by the servicer, a thread for each tenant, which accepts incoming requests from the clients.

```

1  def LoadOverlay(self, request, context):
2      tenant_id = self._get_tenant_id(context)
3      overlay_id, ip_cores = self.resource_manager.load_overlay(
4          tenant_id, request.bitfile_path
5      )

```

The servicer routes the request to the resource manager which will perform two validation steps: (1) verifies the tenant has not exceeded overlay quota by counting active overlays in resources.overlays set, (2) checks bitstream presence in allowed_bitstreams whitelist defined in configuration.

```

1  def load_overlay(self, tenant_id, bitfile_path):
2      with self._lock:
3          if not self.tenant_manager.can_allocate_overlay(tenant_id):
4              raise Exception("Overlay limit reached")
5
6          tenant_config = self.tenant_manager.config.get(tenant_id)
7          if not tenant_config:
8              raise Exception(f"Tenant {tenant_id} not found")
9
10         allowed_bitstreams = tenant_config.allowed_bitstreams or set()

```

After checking the user design's quota and presence, the hypervisor determines which Lake to download the hardware design to. More specifically, the user can choose a specific Lake, if available, through the bitstream's name or leave the selection to the hypervisor. In fact, during the hypervisor's parsing process, in the method `find_best_zone_for_bitstream()` if the bitstream's name in the user's request is preceded by `PR_N_userdesign.bit`, where N is the Lake's number, the hypervisor will attempt to use that exact Lake; otherwise, it automatically selects a free one.

```
1     result = self.pr_zone_manager.find_best_zone_for_bitstream(  
2         bitfile_path,  
3         tenant_id,  
4         self.bitstream_dir,  
5         allowed_bitstreams  
6     )  
7  
8     if not result:  
9         raise Exception(f"No available PR zone for bitstream  
10             {bitfile_path}")  
11  
    zone_id, actual_bitstream_path = result
```

If available, the hypervisor enables the user design download process to the selected Lake using methods from the `dfx_manager` class. This process then decouples the Lake's decoupler, loads the partial bitstream, and re-couples the Lake to the rest of the architecture.

```
1     logger.info(f"[PYNQ] Loading partial bitstream  
2         {actual_bitstream_path} "  
3         f"in PR zone {zone_id} for tenant {tenant_id}")  
4  
    success = self.dfx_manager.reconfigure_pr_zone(zone_id,  
5        actual_bitstream_path)  
6    if not success:  
        raise Exception(f"Failed to reconfigure PR zone {zone_id}")
```

Next, the assignment is registered in the hypervisor's data structures, and the container's access to the virtual device associated with the assigned Lake is enabled.

```
1 uio_device = self._pr_zone_uio_devices.get(0)
2 if os.path.exists(uio_device):
3     os.chown(uio_device, tenant_config.uid, tenant_config.gid)
```

The client then receives the response from the hypervisor. The `uio_device` field allows the client's MMIO class to set up MMIO objects using the virtual device assigned to the tenant (the only one it has permissions for). The response may include metadata for the IP cores contained in the design, similar to the PYNQ library. However, for this latter point, a parser will be included in future work, as Vivado, when working with DFX, loses the metadata in the HWH file where it would normally be.

```
1
2 # Response received by client
3 response = LoadOverlayResponse(
4     overlay_id='overlay_a1b2c3d4',
5     zone_id=0,
6     zone_base_address=0xB0000000,
7     uio_device='/dev/uio10'
8     ip_cores_data={
9         ...
10    }
11 )
```

3.3.3 CMA Buffer management

As previously mentioned, while resource allocation can be handled by the Python hypervisor through RPC channels, buffer memory writes cannot be mediated by this technology due to significant serialization overhead. For this reason, the strategy adopted by μ -VF consists of employing char devices, one per tenant, for accessing CMA buffers with mediation by a low-level kernel module. The system operates across three cooperating levels to establish direct

physical memory access: the Hypervisor (user-space) allocates physical memory through PYNQ CMA allocator and registers authorizations in the kernel module via sysfs interface; a kernel module (kernel-space) maintains per-tenant authorization tables and validates mapping requests, configuring the MMU for address translation, the user container (user-space) executes `mmap()` on the char device, obtaining direct physical memory access through virtual addresses.

Hypervisor-container communication occurs once (during allocation) via gRPC; subsequent data accesses completely bypass hypervisor and kernel, operating through hardware MMU translation.

Listing 3.9: Client requesting a Bitstream through API

```
1 input_buf = allocate(shape=(104,), dtype=np.int32)
```

As illustrate in Figure 3.9, when the client requests a buffer allocation as in the listing 3.9, the request is sent to the hypervisor via RPC. Using mechanisms similar to those we've seen before, the hypervisor checks the resource quotas and, if a new buffer can be allocated, it allows the allocation. For the allocation, the hypervisor uses Pynq's actual APIs, asking the kernel to allocate a contiguous memory buffer. Once the memory is allocated, the kernel will return the physical address to the hypervisor and will configure the Lake's PL-MMU as discussed in section 2.4.2.

Listing 3.10: Extract of the Hypervisor buffer manager

```
1 def allocate_buffer(self, tenant_id, shape, dtype):
2     size = np.prod(shape) * np.dtype(dtype).itemsize
3     # Verify tenant quota
4     if not self.tenant_manager.can_allocate_buffer(tenant_id, size):
5         raise Exception("Buffer allocation limit reached")
6     # Allocate from CMA region
7     buffer = pynq.allocate(shape=shape, dtype=dtype)
8     physical_address = buffer.physical_address
```

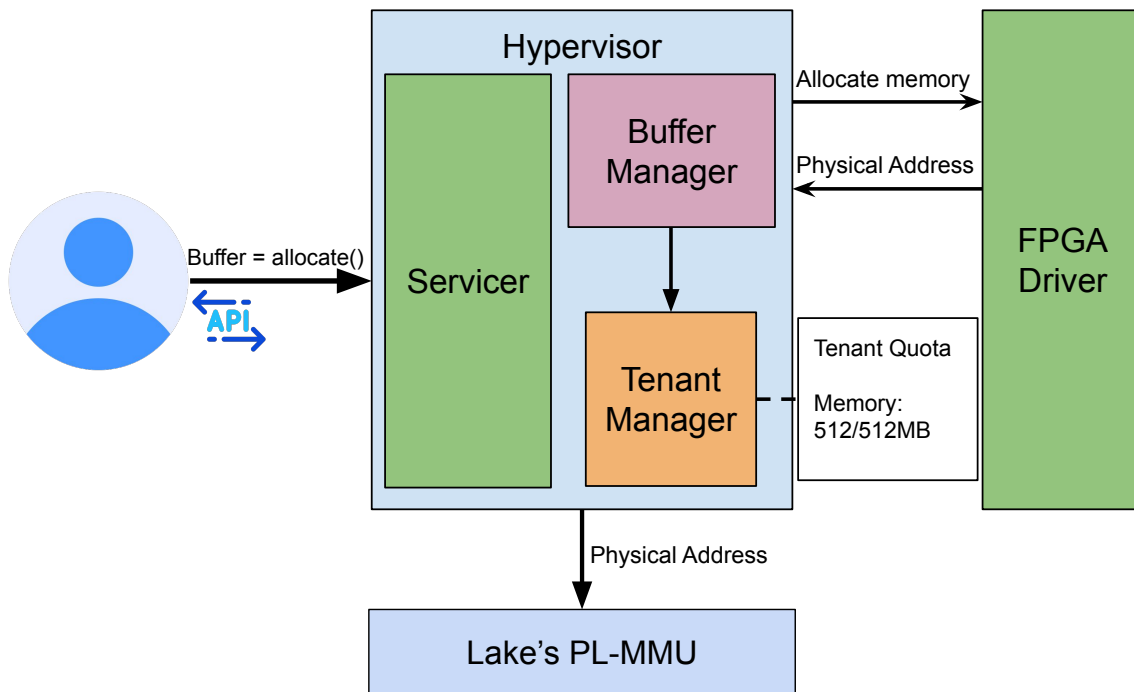


Figure 3.9: High-level overview of the buffer allocation request flow, showing the interaction between the Client API, the Hypervisor’s Buffer and Tenant Managers, and the underlying FPGA Driver.

To enable controlled access, the hypervisor registers the buffer in the kernel module’s authorization table by writing to sysfs.

Listing 3.11: Extract of register_buffer_in_char_device method of the Buffer Manager

```

1 #Hypervisor
2 def _register_buffer_in_char_device(self, tenant_id: str, buffer_id:
   str, phys_addr: int, size: int) -> int:
3 # ...
4     command = f"{vm_offset:x},{phys_addr:x},{size:x},{buffer_id}"
5     try:
6         with open(sysfs_path, 'w') as f:
7             f.write(command + '\n')

```

When the hypervisor writes to the sysfs interface, the kernel module’s add_buffer_store()

callback is invoked. This function parses the command string and creates an authorization entry in the tenant's table. The overall flow of this operation is shown in Figure 3.10, while the corresponding data structure population is detailed in Listing 3.12.

Listing 3.12: Extract of `add_buffer_store` function of the low-level driver

```
1 mapping = kzalloc(sizeof(*mapping), GFP_KERNEL);
2 mapping->vm_offset = offset;
3 mapping->phys_addr = phys_addr;
4 mapping->size = size;
5 mutex_lock(&tenant->lock);
6 list_add_tail(&mapping->list, &tenant->buffers);
7 }
```

The command string received from the hypervisor is parsed using `sscanf()` to extract four components: virtual offset, physical address, size, and buffer identifier. A `buffer_mapping` structure is then allocated in kernel memory via `kzalloc()` and populated with the parsed data. After the new mapping is appended into tenant data structure, the tenant's authorization table contains a new entry that will be consulted during subsequent `mmap()` operations to validate access requests.

The virtual offset serves as a unique buffer identifier for that tenant. The increment equals the buffer size rounded up to the next page boundary (4KB) to prevent overlaps when multiple buffers are mapped into the container's virtual address space. For multiple buffers, multiple entries are registered in the kernel's authorization table, each with a distinct virtual offset. The container accesses different buffers by specifying different offsets in separate `mmap()` calls. The kernel module validates each request; the container cannot modify the authorization table (only the hypervisor, running as root, can write to `sysfs`).

Once the hypervisor has registered buffer authorizations in the kernel module, the container receives the authorization parameters via gRPC response: the `vm_offset` (the authorized offset for `mmap`), and the `char_device_path` identifying the tenant's device node.

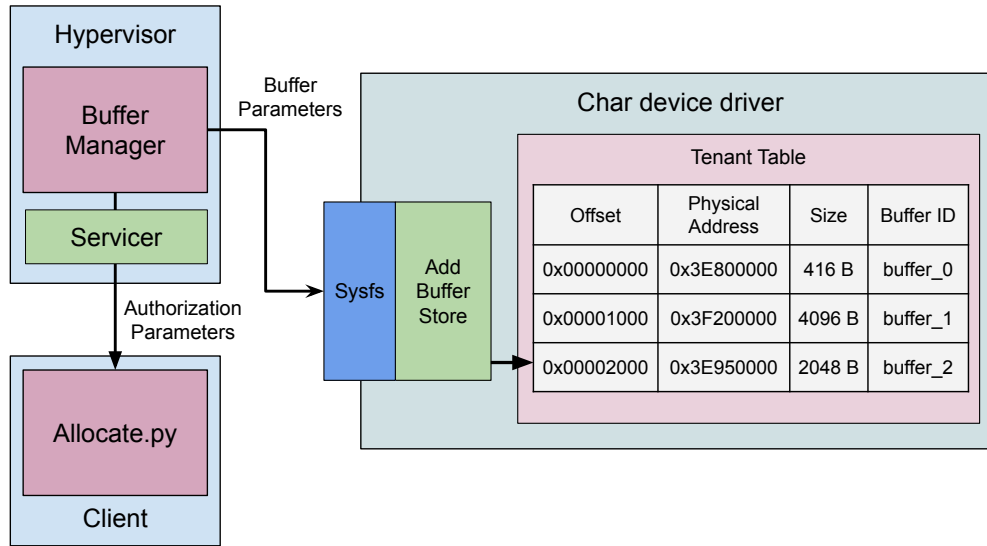


Figure 3.10: Buffer registration mechanism. The Hypervisor performs two operations: (1) sends buffer parameters back to the client via the Servicer, and (2) sends authorization parameters to the char device driver via the sysfs interface, which updates the internal Tenant Table with access permissions.

The container establishes direct memory access by executing `mmap()` on its char device with the authorized offset:

Listing 3.13: Container-side zero-copy memory mapping

```

1 class ProxyBuffer:
2     def _setup_char_device(self, device_path, vm_offset, shape,
3                             dtype):
4         self._char_mmap = mmap.mmap(
5             fileno=self._char_fd,
6             length=size,
7             flags=mmap.MAP_SHARED,
8             prot=mmap.PROT_READ | mmap.PROT_WRITE,
9             offset=vm_offset # Validated by kernel module
10        )

```

When a container executes `mmap()`, the kernel invokes the char device's `mmap` callback,

which validates authorization and configures the PS-MMU for physical memory access.

Listing 3.14: Authorization validation

```
1 unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
2 mutex_lock(&tenant->lock);
3 mapping = find_mapping_by_offset(tenant, offset);
4
5 if (!mapping || size > mapping->size) {
6     return -EACCES;
7 }
```

The function searches the tenant's authorization table for the requested offset. If not found or if the requested size exceeds the buffer size, the operation fails.

Listing 3.15: MMU configuration

```
1 ret = remap_pfn_range(
2     vma,
3     vma->vm_start,
4     mapping->phys_addr >> PAGE_SHIFT,
5     size,
6     vma->vm_page_prot
7 );
```

If authorization succeeds, `remap_pfn_range()` configures the PS-MMU's page tables to map the container's virtual addresses directly to the authorized physical CMA address. After this configuration, the PS-MMU automatically translates all subsequent memory accesses without kernel intervention. With this mechanism, numpy array operations translate to direct memory loads and stores at the physical addresses configured by the kernel module during `mmap()` with data going directly to physical CMA memory without serialization. The overall mechanism described above is illustrated in Figure 3.11.

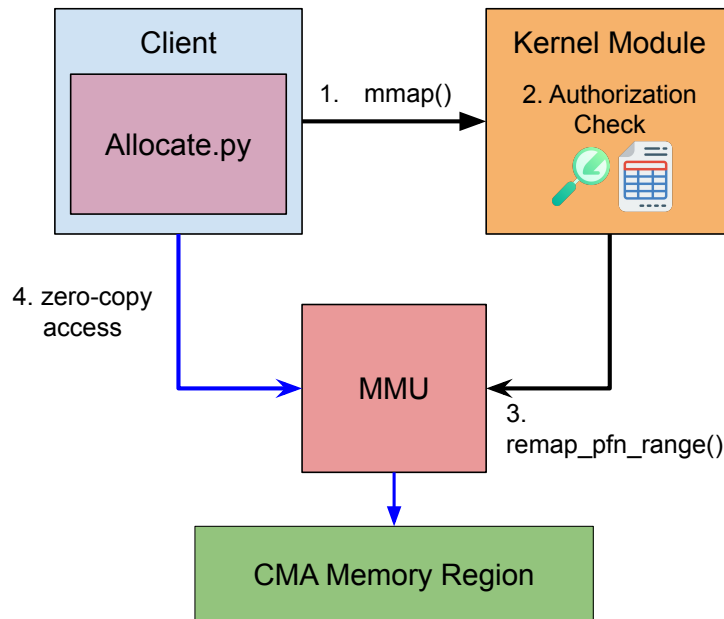


Figure 3.11: Zero-copy access mechanism. The client's `mmap()` call is validated by the Kernel Module, which then uses `remap_pfn_range()` to configure the PS-MMU, providing the client with direct access to the physical CMA memory region.

3.3.4 MMIO management

For MMIO region access, the system employs UIO (Userspace I/O), where each reconfigurable lake corresponds to a separate UIO device (`/dev/uioX`), with access control managed through standard Linux file permissions. This differs from CMA buffer management, where dynamic allocation requires kernel module mediation. MMIO regions have fixed physical addresses known at boot time, making static device tree definitions appropriate.

Virtual devices are defined using device tree overlay files. Each overlay specifies a UIO (Userspace I/O) device with a constrained address range matching the associated reconfigurable Lake.

Listing 3.16: Device tree entry for a virtual MMIO device

```

1 compatible = "xlnx,zynqmp";
2 fragment@0 {
3     target-path = "/";

```

```
4      __overlay__ {  
5          pr_0@b00000000 {  
6              compatible = "generic-uio";  
7              reg = <0x0 0xb0000000 0x0 0x00010000>;  
8              no-interrupts;
```

The `reg` property defines the accessible address range for the virtual device. In the example of listing 3.16, the device `pr_0` can access physical addresses from `0xb0000000` to `0xb000ffff` (64 KB range), corresponding to the MMIO registers of reconfigurable lake 0. This address restriction ensures that the virtual device cannot access memory regions belonging to other lakes or system resources. Multiple virtual devices are created by defining additional overlays with distinct address ranges.

After the compilation, the overlays are loaded at system initialization using PYNQ's `DeviceTreeSegment` API:

Listing 3.17: Loading device tree overlays via PYNQ

```
1 from pynq.devicetree import DeviceTreeSegment  
2 dt_segment_pr_0 = DeviceTreeSegment("/home/xilinx/PR_0.dtbo")  
3 dt_segment_pr_1 = DeviceTreeSegment("/home/xilinx/PR_1.dtbo")  
4 dt_segment_pr_2 = DeviceTreeSegment("/home/xilinx/PR_2.dtbo")  
5 dt_segment_pr_3 = DeviceTreeSegment("/home/xilinx/PR_3.dtbo")  
6 dt_segment_pr_0.insert()  
7 dt_segment_pr_1.insert()  
8 dt_segment_pr_2.insert()  
9 dt_segment_pr_3.insert()
```

At this point, the virtual devices are loaded in the device tree and can be mounted into user containers. As seen in section 2.4.2, access to the authorized UIO device is granted by the hypervisor at the time of user bitstream loading.

The container accesses MMIO registers through a MMIO wrapper class that interfaces with the assigned UIO device:

Listing 3.18: Client-side MMIO access via UIO device

```
1 class MMIO:
2     def __init__(self, uio_path, length):
3         self.fd = os.open(uio_path, os.O_RDWR | os.O_SYNC)
4         self.mmap = mmap.mmap(self.fd, length, mmap.MAP_SHARED,
5                               mmap.PROT_READ | mmap.PROT_WRITE)
6         self.array = np.frombuffer(self.mmap, dtype=np.uint32)
7     def write(self, offset, data):
8         idx = offset >> 2
9         self.array[idx] = np.uint32(data) # Direct MMIO write
10    def read(self, offset):
11        idx = offset >> 2
12        return int(self.array[idx]) # Direct MMIO read
```

CHAPTER

4 | Evaluation

In this section, we evaluate the performance of the proposed virtualization framework with a focus on the overhead introduced by multi-tenancy and hardware abstraction. Since μ -VF does not interfere with, or modify, the compute logic implemented by tenants, the evaluation targets only the components affected by the virtualization stack, namely: the resource overhead of the hardware stack, the latency and throughput of memory and I/O interactions, and the responsiveness of dynamic GPIO reconfiguration.

4.1 Experimental Platform

All experiments were conducted on an AMD Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit, which integrates a quad-core ARM Cortex-A53 Processing System (PS) alongside a programmable logic (PL) fabric. The system ran PYNQ v2.7.0, which provides a Python-based runtime for interacting with programmable logic resources.

Hardware designs were developed and synthesized using Vivado™ Design Suite 2024.2, while the software components, including the μ -VF hypervisor and container orchestration, were implemented using a custom runtime built on top of the standard PYNQ stack. Unless otherwise stated, all measurements were performed with a configuration supporting up to 4 concurrent tenants, 32 virtual GPIO pins, and 8 physical I/O pins. This setup was chosen to reflect a realistic multi-tenant deployment on resource-constrained embedded platforms.

Table 4.1: FPGA resources available on the AMD Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit

Resource	Quantity
Look-Up Tables (LUTs)	274,080
Flip-Flops (FFs)	548,160
DSP Slices	2,520
Block RAM (BRAM) Tiles	912
UltraRAM (URAM) Blocks	320
CLBs (Configurable Logic Blocks)	42,000

4.2 Programmable Logic resource overhead

The resource overhead introduced by the virtualization layer becomes particularly critical in embedded FPGA systems, where the availability of logic resources is significantly more constrained compared to data center-class FPGAs (see Table 4.1 for the resource breakdown of the ZCU102). To address this, we designed the hardware stack to minimize its footprint, thereby maximizing the amount of logic resources that can be allocated to user-defined regions. Data transfers between the programmable logic and DDR memory are performed over a 128-bit AXI interface, which represents the maximum data width supported by the ZCU102's high-performance (HP) ports. In contrast, MMIO-based register accesses use a 32-bit AXI-Lite interface, consistent with standard control path configurations.

Figure 4.1 reports the pre-implementation resource utilization of the hardware stack (Shore+ATLAS). The results show that the hardware stack occupies less than 10% of the available LUTs, LUTRAMs, and flip-flops, while requiring no BRAM or DSP blocks. Most of the resource consumption is attributed to the shared interconnect bus, whereas ATLAS, even when configured to expose 8 physical pins, contributes negligibly to the overall footprint. As a result, the resource utilization remains nearly constant as the number of exposed physical pins increases.

Table 4.2 reports the post-implementation allocation of logic resources to each Lake. The resource assignment was guided by the principle of fair distribution among tenants, while simultaneously aiming to maximize the overall resource utilization. This was achieved by minimizing the footprint of the hardware stack, as can be seen from the floorplanning view in

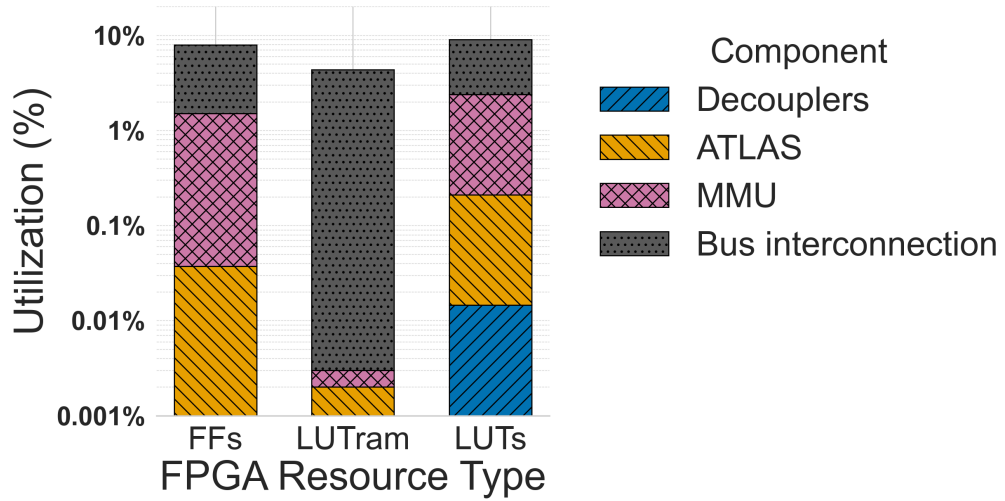


Figure 4.1: Hardware stack resource usage breakdown after synthesis. Components include decouplers for electrical isolation, ATLAS for GPIO virtualization, MMUs for memory protection, and bus interconnection fabric. Total utilization remains below 10% for all resource types

Table 4.2: Resource allocation per Partial Region (PR) on ZCU102. ”% FPGA” indicates the percentage of total FPGA resources, while ”% Lakes” shows the percentage relative to the total resources available for Lakes (85% of the FPGA).

Region	LUTs		LUTram		BRAM		DSP	
	% FPGA	% Lakes	% FPGA	% Lakes	% FPGA	%Lakes	% FPGA	% Lakes
Lake 1	18.39%	20.40%	18.75%	20.45%	19.74%	21.05%	21.43%	23.08%
Lake 2	22.99%	25.50%	23.75%	25.91%	19.74%	21.05%	25.00%	26.92%
Lake 3	18.26%	20.27%	17.25%	19.61%	19.74%	21.05%	14.29%	15.38%
Lake 4	25.83%	33.83%	25.81%	34.03%	26.85%	36.84%	25.00%	34.62%

Figure 3.6, ensuring sufficient space for its placement and enabling congestion-free routing, an especially challenging task due to the heterogeneous and fragmented layout of the ZCU102 device.

Overall, the post-implementation allocation reserves approximately 85% of the total available resources for user workloads. This represents a significant improvement over prior approaches [27] on the same ZCU102 platform, where only around 50% of the fabric could be effectively allocated to tenant workloads due to the footprint of the virtualization layer and routing constraints.

4.3 MMIO latency

As discussed in Section 2.4.2, particular attention was devoted to minimizing the latency of user-level memory-mapped I/O (MMIO) accesses. We evaluated the effectiveness of our virtualization approach by analyzing two key scenarios: (i) a single-tenant configuration, in which we compared the access latency of an application running in the native environment versus the latency observed when the same application is executed through the full μ -VF stack, including containerization and MMIO virtualization; (ii) a multi-tenant scenario, in which several tenants concurrently access their respective memory-mapped registers.

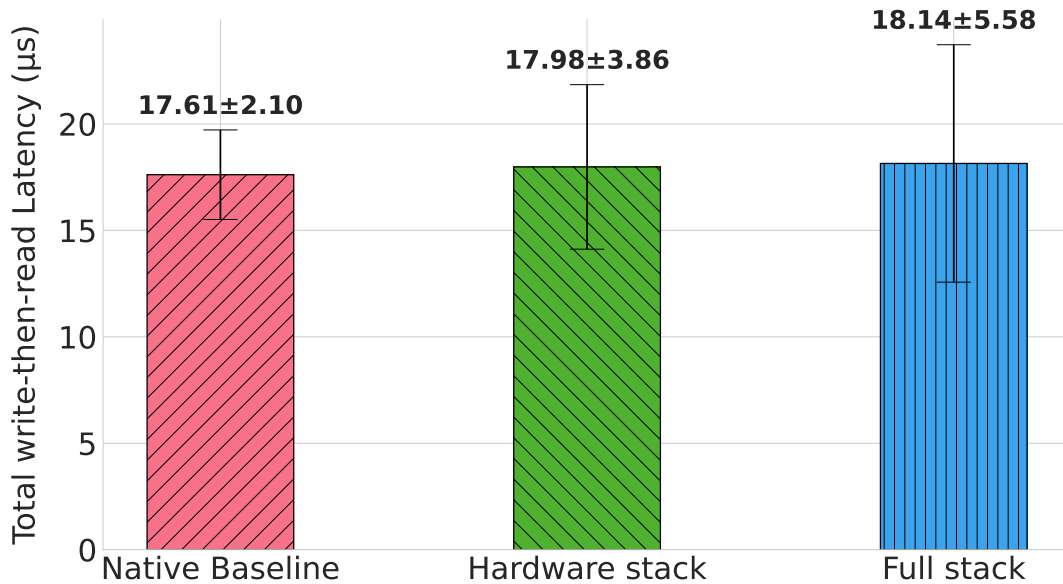


Figure 4.2: MMIO access latency comparison between native (non-virtualized) execution and μ -VF with incremental virtualization layers in single-tenant mode. Measurements show: native baseline, hardware stack only (Shore + ATLAS), and full μ -VF stack (hardware + software virtualization).

All reported values represent the average of 10,000 read-then-write cycles targeting tenant-specific MMIO registers inside Lakes. Measurements were taken from within user-space processes running inside isolated containers when evaluating the full software stack overhead, and from native processes when measuring the hardware baseline.

Figure 4.2 compares the average access latency for a complete write-then-read cycle across three configurations: (i) a native setup with a single hardware design and no virtualization,

(ii) the same hardware design integrated within the hardware stack, and (iii) the full μ -VF stack, including the hardware and software stack. In this single-tenant scenario, the results demonstrate that the hardware stack introduces an overhead of approximately 2.06%, while the complete virtualization stack adds a total overhead of 2.93% compared to the bare-metal baseline. These minimal overheads validate our architectural decisions: the hardware stack's low overhead (2.06%) confirms that the Shore infrastructure and bus interconnect add negligible latency to the AXI-Lite path. The additional 0.87% introduced by the software stack demonstrates the effectiveness of our zero-copy approach: containers access hardware registers directly through memory-mapped device files without continue hypervisor mediation. To put this in perspective, if we had adopted a traditional Remote Procedure Calls (RPCs)-based approach for MMIO access, our measurements indicate that each operation would incur approximately 80 μ s of overhead, more than 4.4 \times the total latency of our zero-copy implementation.

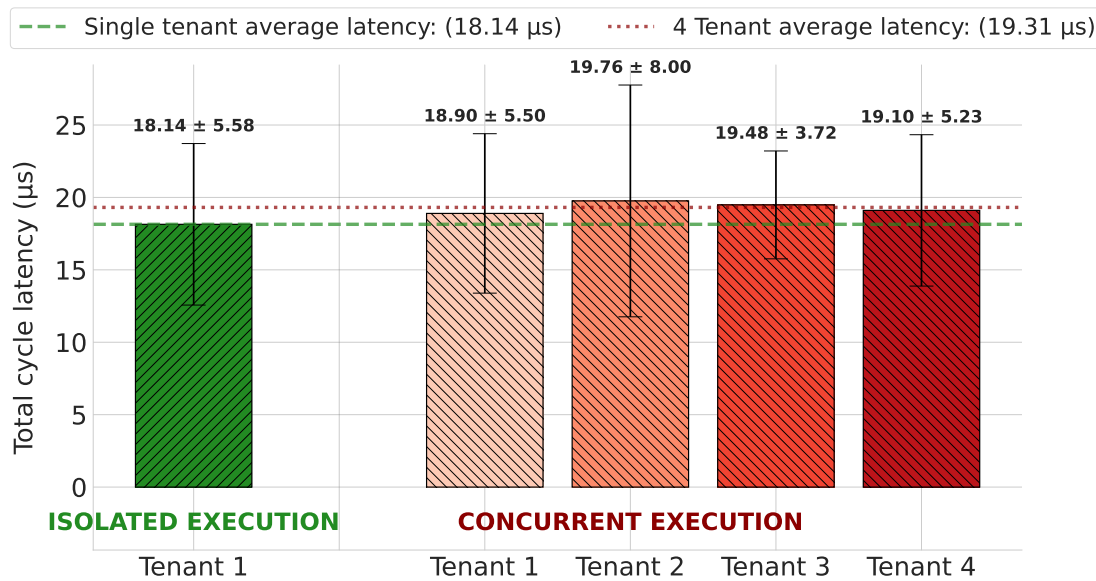


Figure 4.3: MMIO access latency under isolated and concurrent execution with full μ -VF virtualization (hardware + software stack). The graph shows the average write-then-read cycle latency for a single tenant running alone (isolated execution) versus four tenants simultaneously performing read/write operations to their respective Lake registers (concurrent execution).

Figure 4.3 reports the average latency for a full write-then-read cycle to MMIO registers under both isolated and concurrent execution scenarios. In the isolated case, a single tenant

runs alone on the system in a Lake, while in the concurrent setup, all four tenants operate simultaneously on distinct Lakes.

Despite concurrent usage, the system maintains bounded latency. The average access latency increases only modestly, from 18.14 μs (single tenant) to 19.31 μs (multi-tenant average), corresponding to a 6.5% increase. This demonstrates that the $\mu\text{-VF}$ stack scales effectively across tenants, without introducing significant contention or delay in MMIO access.

4.4 Memory throughput

In FPGA environments, maintaining high memory throughput is critical to ensure that application accelerators can efficiently access shared DRAM without becoming bottlenecked by the virtualization layer. Since user logic frequently exchanges data with the processing system, any overhead introduced by the hardware stack may limit performance scalability, particularly in real-time or data-intensive edge deployments. To evaluate the impact of $\mu\text{-VF}$'s hardware stack on raw memory bandwidth, we benchmarked AXI4 burst transactions initiated directly by user logic mapped to a tenant Lake. To quantify throughput, we instrumented the system with an on-chip AXI Timer, placed inside a Lake and controlled via AXI-lite registers. For write operations, the timer captures the number of clock cycles elapsed from the assertion of AWVALID (start of address phase) to the handshake completion signaled by BVALID & BREADY. This interval accurately includes the full latency of the AXI protocol, arbitration, shell routing, and any buffering or interconnect overhead.

Our benchmarks compare the native baseline, the $\mu\text{-VF}$ virtualization stack with one tenant, and a multi-tenant configuration with four active tenants on dedicated Lakes. In the single-tenant configuration, as per the graphs in Figure 4.4, all transactions were routed through the AXI HP0 port, a 128-bit interface operating at 100 MHz on the ZCU102 board. This setup was used to isolate and evaluate the impact of the virtualization stack itself, independent of inter-tenant interference. In contrast, in the multi-tenant experiments, each Lake was assigned a dedicated HP port (HP0–HP3), enabling concurrent memory access while preserving bandwidth isolation.

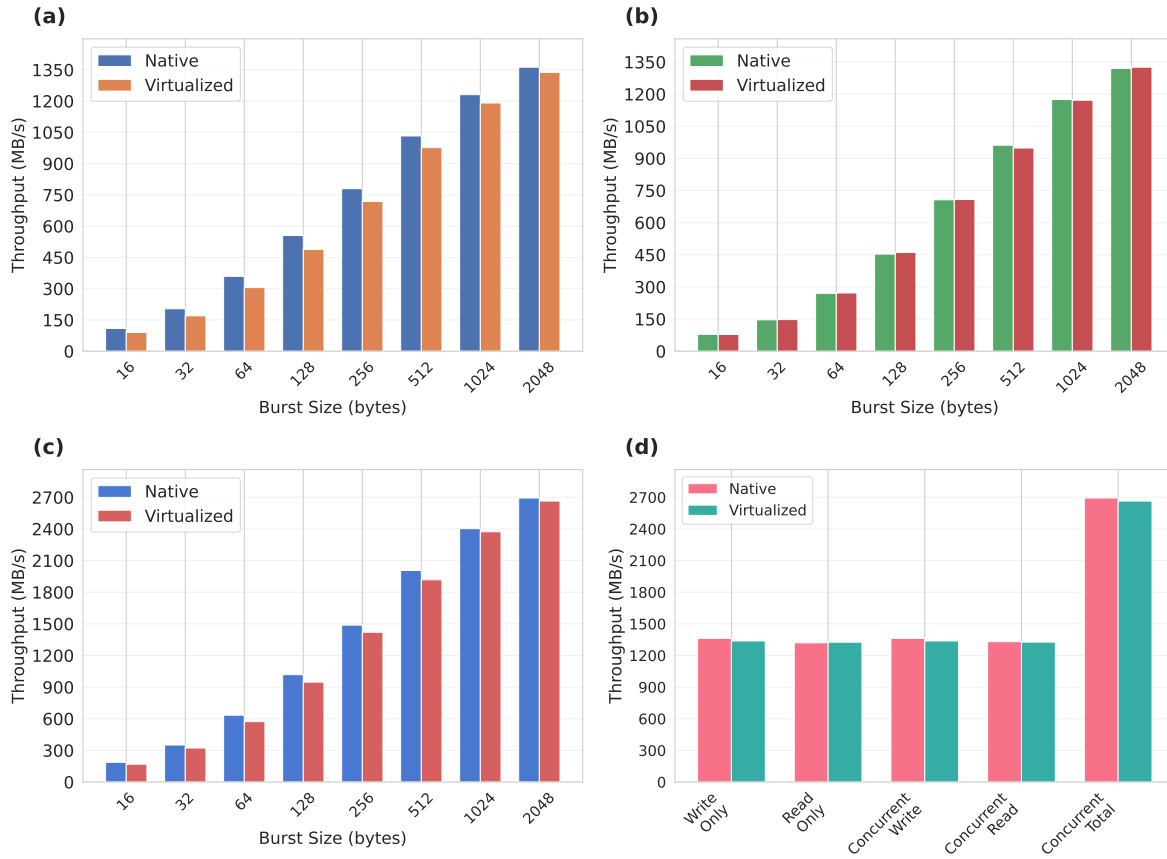


Figure 4.4: Single-tenant memory throughput on HP0 port: native vs virtualized execution. (a) Write throughput, (b) Read throughput, (c) Concurrent R+W aggregate throughput across burst sizes from 16B to 2KB. (d) Summary at 2048B burst size.

In write-only tests, as shown in Figure 4.4(a), throughput dropped slightly from 1362.6 MB/s (native) to 1337.8 MB/s with the shell, corresponding to a modest 1.8% overhead. We attribute this degradation to the presence of the MMU, which verifies AXI write addresses (AWADDR) against secure memory regions at runtime. Similarly, read-only throughput 4.4(b) remained effectively unchanged, increasing marginally from 1320.0 MB/s to 1325.7 MB/s (+0.4%), falling within the expected range of DRAM controller variability.

When performing concurrent reads and writes, total throughput 4.4(c) decreased from 2693.4 MB/s to 2664.4 MB/s (1.1%). As burst size increases, the fixed cost of address and control signaling is distributed over more data, while the AXI bus and memory controller can operate more efficiently in a pipelined fashion, resulting in higher sustained throughput.

In Figure 4.5 we report throughput in the worst case scenario, i.e., the condition in which

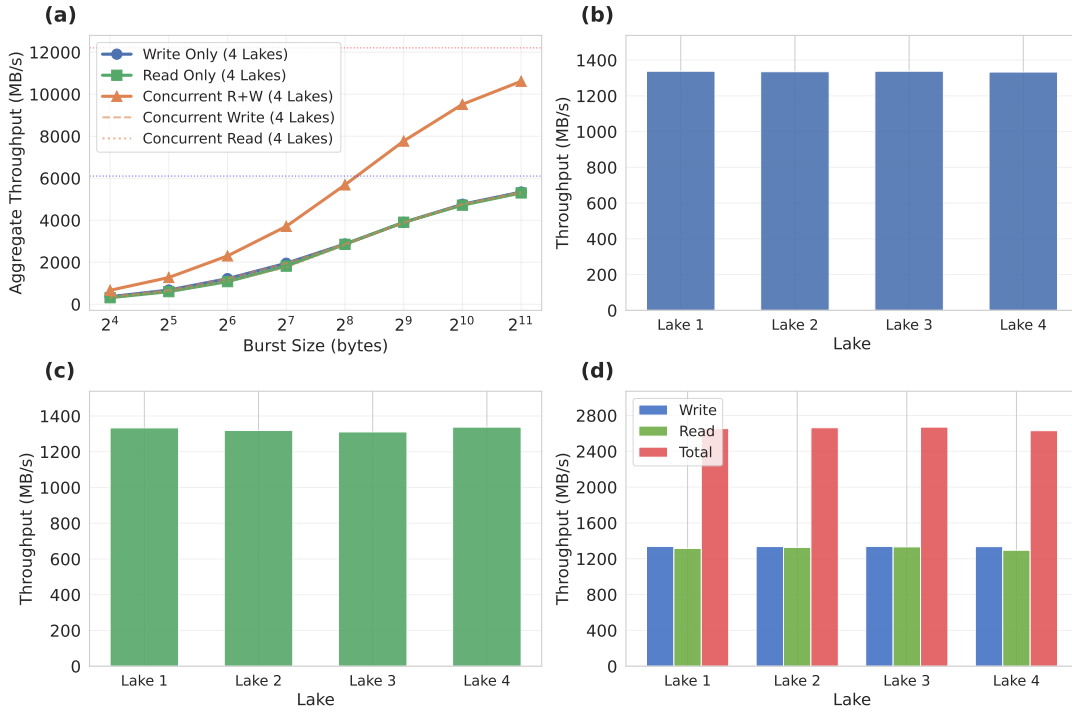


Figure 4.5: Multi-tenant memory throughput evaluation with four concurrent tenants. (a) Aggregate throughput scaling across burst sizes. (b) Write throughput distribution per Lake at 2048B burst size. (c) Read throughput per Lake showing balanced performance. (d) Concurrent read/write throughput demonstrating 10,616 MB/s aggregate bandwidth across all tenants, with each Lake assigned to a dedicated HP port.

all tenants concurrently access the DDR. As visible from the graphs, the bandwidth is equally divided among the Lakes with slight differences due to the different HP ports assigned, one per tenant (Lake). The overhead compared to the single-tenant case remains minimal: as shown in Figure 4.5(b) each Lake achieves an average of 1335.9 MB/s for write operations (compared to 1337.8 MB/s in single-tenant mode) and, as per Figure 4.5(c) 1325.2 MB/s for reads (versus 1325.7 MB/s single-tenant). This demonstrates that the multi-tenant configuration introduces negligible additional overhead beyond the base virtualization cost. When all four tenants perform concurrent read and write operations, Figure 4.5(d) shows that we achieve an aggregate throughput of 10,616 MB/s, representing a 17.1% improvement over previous approaches [27].

4.5 ATLAS evaluation

ATLAS employs a two-stage synchronization architecture to ensure atomic and glitch-free GPIO reconfiguration. The remapping latency can be analytically derived by examining the four distinct phases of the reconfiguration process, two of which are synchronous and two combinational. The first stage captures the new configuration written via the AXI interface within a single clock cycle. This is followed by a combinational phase in which the new mapping is decoded and validated. The logic extracts pins ownership information and checks for conflicts (i.e., two virtual pins connected to the same physical pin). Although the combinational phase introduces a real propagation delay, it is designed to complete within the timing budget of a single clock period and thus does not incur an additional clock cycle from a pipeline perspective. To prevent potential glitches during reconfiguration, the outputs of the combinational logic are latched by a second synchronization register before being routed to the physical I/O pads. The final routing to the physical pads is again combinational and incurs negligible additional delay and thus the observable latency after writing a new configuration is two clock cycles. At the operating frequency of 100 MHz, this corresponds to a reconfiguration delay of 20 ns, regardless of the number of pins involved. This latency refers solely to the internal ATLAS remapping pipeline and is sufficiently low to enable fine-grained time-multiplexing of physical I/Os. For instance, configurations can be preloaded into a BRAM, which replaces AXI as the source of configuration data, and applied in two clock cycles, enabling time-sharing strategies where the mapping configuration changes periodically. When configuration updates are issued under the control of the hypervisor through AXI, the total latency includes the AXI write operation. As reported in Figure 4.4, the measurement corresponding to access performed outside the containerized environment, reflecting the hardware-stack-only contribution relevant to the hypervisor, adds approximately 18 s to the reconfiguration time. When accounting for the additional 2 clock cycles of the latency introduced by ATLAS, the total time to apply a new configuration is approximately 18.02 s at 100 MHz.

4.5.1 Request-to-Ready latency

To provide a comprehensive evaluation of our system’s responsiveness and overall user experience, we conducted a series of experiments to measure two key end-to-end latency metrics. The first, Lake Reconfiguration Latency (warm-request), isolates the time required to configure and make the Lake available after the user’s container has already been successfully started. The second metric, End-to-End Latency (cold-request), measures the total time from a user’s initial request until a Lake is fully configured and ready for use. This includes all overheads, most notably the container startup time. While this reconfiguration time can depend on the size of the pblock, our experiments used similarly sized pblocks, which ensures that the measurements provide a consistent basis for comparison. To simulate realistic resource usage and the presence of an increasing number of tenants, we performed these measurements under different CPU utilization levels: idle, light CPU load (50%), and heavy CPU load (90%). For each condition, we collected 50 samples to ensure statistical significance.

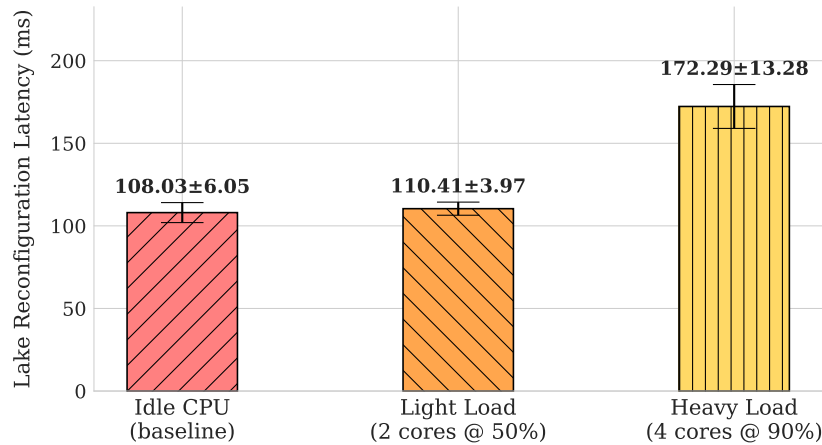


Figure 4.6: Lake Reconfiguration Latency (Warm-Request) vs. CPU Load. The time measured includes the client’s API request, bitstream allocation, and hypervisor response with metadata.

The first set of results, shown in Figure 4.6, measures the time it takes for a user to gain access to an allocated Lake when the environment is already running. This warm-request measurement begins the moment the client uses our API to request a bitstream allocation and concludes when the hypervisor responds with a success notification and returns the necessary metadata. As the graph illustrates, the influence of the CPU load is negligible between the idle

baseline and the 50% load condition, with the latency increasing by only approximately 2 ms (from 108.03 ± 6.05 ms to 110.41 ± 3.97 ms). However, the effect becomes more relevant under a heavy 90% CPU load, where the latency rises to 172.29 ± 13.28 ms.

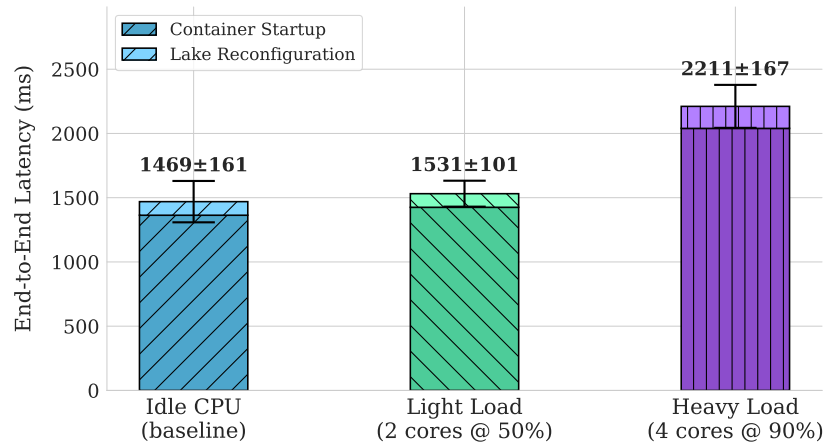


Figure 4.7: End-to-End Latency (Cold-Request) Breakdown vs. CPU Load. Total time includes Container Startup and Lake Reconfiguration.

Figure 4.7 illustrates the cold-request end-to-end measurements. The total time measured for a cold request is comprehensive, including the container startup (with the virtual devices mounted), the launching of the client process, the client's request for a bitstream, and the final allocation of the Lake. Similar to the warm-request results, the influence of the CPU load is minimal in the case of 50% utilization, but becomes more tangible with heavy 90% loads. However, comparing this graph to the warm-request latency highlights that the total end-to-end time is dominated by the container startup overhead, which is a significant factor in the overall user experience.

CHAPTER

5

Related Work

This chapter analyzes the state of the art of FPGA virtualization, examining the frameworks proposed in the literature. The analysis will focus on their suitability for the domain of embedded systems and edge computing, in order to identify the limitations of existing solutions and motivate the contribution of μ -VF.

5.1 Approaches for Multi-Tenancy and Resource Optimization

One of the first approaches to FPGA virtualization in the cloud was proposed by Byma et al. [6], where a Xilinx Virtex 5 was integrated into an IaaS Cloud using OpenStack. Through the use of DPR, the FPGA was partitioned into Virtualized FPGA Resources (VFRs), each intended for a user. Communication with these partitions, each of which has a fixed size, occurs through network interfaces, managed by a static shell that also deals with access to DDR memory. Although innovative for its time, three fundamental limitations emerge for the application of this work in embedded systems. First, the entire architecture depends on an external infrastructure. The management and reconfiguration of the VFRs are delegated to a software agent hosted on an external host, not respecting the requirement of stand-alone operation of edge devices. Second, the framework focuses exclusively on the virtualization of computing resources with network and memory interfaces. The virtualization of the GPIO is

completely absent from the model.

Later works have focused on optimizing resource usage and the interaction between software on x86 hosts and accelerators on FPGAs. AmorphOS [11] is an open-source operating system for PCI-E FPGAs. It introduces the concept of a Morphlet, an abstraction that encapsulates the user's logic and allows the system to scale and dynamically reposition it on the fabric based on available resources, solving the problem of fragmentation, due to the partitioning of the FPGA with fixed-size regions [6]. The architecture of AmorphOS is articulated on two main components: a software stack, running on an external x86 host, and a hardware shell on the FPGA. The software stack provides the set of APIs and drivers for interaction, as well as a scheduler and a resource manager (zone-manager) for the assignment of Morphlets. The hardware shell, on the other hand, deals with virtualizing access to low-level resources, such as the PCIe bus, DDR memory, and standard I/O interfaces.

ViTAL [31] and subsequently Hetero-ViTAL [30] focus on optimizing the use of resources on an FPGA cluster. In this model, a custom compiler, but based on Vivado, breaks down the user's design into its elementary logic modules (at the VHDL/Verilog level). These "virtual micro-blocks" are then mapped and combined in a flexible way on the fabric, creating an acceleration region of a size close to what is necessary, thus reducing the waste of resources.

Works subsequent to those analyzed have introduced even more advanced techniques, such as overhead optimization through Network-on-Chip (NoC) architectures [17] and memory virtualization systems based on virtual addresses and pagination [12][14][32]. It is fundamental to note, however, that all these frameworks share a common set of assumptions that define their scope and limits. They were designed for large FPGAs connected via PCIe, where the primary interaction channel is the PCIe bus and the entire software stack is managed by a powerful external x86 host, while the GPIO, not of fundamental interest in such areas, was not virtualized.

5.2 Overlay Architectures for Bitstream Portability

Parallel to the efforts to enable multi-tenancy, another important line of research has addressed the problem of bitstream portability. A bitstream is, by its nature, strictly linked to the physical architecture of a specific FPGA model, making a design not transferable to different devices. A solution to this problem is represented by overlay architectures, which introduce a virtual abstraction layer above the physical fabric. More specifically, an overlay architecture is itself the description in HDL of an FPGA architecture, complete with virtual primitives such as LUTs, flip-flops, and interconnection matrices, which is synthesized on the physical FPGA. The user's design is then compiled using the primitives of this virtual architecture as a target, generating a "virtual bitstream" that is portable on any device capable of hosting the same overlay.

However, this abstraction entails a notable cost in terms of overhead. One of the first proposed overlay architectures [13] required about 100 physical LUTs for every single virtual LUT exposed to the user. Although it demonstrated the feasibility of the concept, such an overhead was unsustainable for practical applications. Years later, the Zuma framework [4] managed to significantly reduce this cost, bringing it to about 40 physical LUTs per virtual LUT, but maintaining a strong functional limitation, such as support for only combinational circuits. More recently, the work of Najem et al. [18] extended the Zuma architecture to also include sequential circuits. However, the spatial overhead remains high, while the performance overhead, due to the low supported clock frequencies (in the order of a few MHz), makes the architecture not very suitable for complex applications.

Given the importance of portability in an IaaS context, during the initial phases of this thesis work, the adoption of an overlay architecture was seriously considered. Prototype versions were developed in which an existing architecture was modified to support programming by the Processing System and interfacing on the AXI bus.

Experimental verifications on these prototypes revealed an unexpected advantage in terms of reconfiguration speed. Through empirical tests, a time of only 0.5 ms was measured to configure a 1000 virtual LUT overlay, compared to the 50 ms required by standard Partial

Reconfiguration for an area of comparable dimensions.

However, despite this significant advantage in configuration latency, the unsustainable resource overhead and severe performance limitations, discussed previously, led to the decision to discard this approach in favor of a model based on native Partial Reconfiguration, which guarantees maximum efficiency for the final applications of the tenants.

5.3 Virtualization on SoC-FPGA Platforms

While the virtualization of FPGAs for the data center represents a mature and widely documented research area in the literature, a notably lower attention has so far been paid to the virtualization of SoC-FPGA platforms, despite their growing importance in the edge computing domain.

Among the most relevant works and most similar to μ -VF are FOS (FPGA Operating System) [27] and Ker-ONE [28], which are examined and discussed below.

5.3.1 Analysis of FOS

FOS is one of the most relevant works in the field of virtualization for SoC-FPGAs and shares with μ -VF the goal of providing a modular operating system for dynamic workloads. Similarly to μ -VF, FOS partitions the Programmable Logic (PL) into a static "shell" and multiple partially reconfigurable regions, managed by a hypervisor (or "daemon") that operates on-device on the SoC's ARM processor.

Despite these architectural similarities, fundamental differences emerge in the virtualization model offered by the two frameworks. In the Task-Based model of FOS, users send hardware tasks to the framework that the hypervisor schedules and executes for them on the available regions. This means that the user never obtains exclusive control of a persistent virtual instance, losing the abstraction of owning a dedicated device. This approach differs sharply from that of μ -VF, which instead assigns each user a persistent vFPGA, that is, a complete virtual instance with isolated resources on both the PS (container) and the PL.

Another fundamental difference lies in software-level isolation. In the "task-based" model

of FOS, each tenant interacts via APIs with a shared system daemon, lacking a system-level isolation mechanism such as containers. This software isolation is instead a fundamental requirement for the persistent vFPGA model of μ -VF. The abstraction of a "Virtual FPGA," in fact, is not limited to providing the user with a private portion of the fabric, but must necessarily also include a private and isolated Processing System, to allow each tenant to execute their software stack securely and independently and tightly coupled to their private fabric region.

Furthermore, FOS does not address the virtualization of I/O resources. Its architecture does not provide mechanisms for the dynamic sharing of GPIO pins between multiple tenants, limiting its applicability in edge and IoT scenarios where interaction with sensors and actuators is fundamental. In contrast, μ -VF introduces ATLAS, a dedicated hardware layer specifically to fill this gap.

Finally, a direct comparison of resource efficiency, conducted on the same ZCU102 platform, highlights a further limitation. As reported by the FOS authors, their framework manages to allocate only about 50% of the device's logical resources to tenants. As will be demonstrated quantitatively in Chapter 4, μ -VF significantly improves this aspect, bringing the available area for users up to 85%.

5.3.2 Analysis of Ker-ONE

Another relevant framework for SoC-FPGAs is Ker-ONE, a hypervisor focused on managing reconfigurable accelerators with real-time constraints. The approach of Ker-ONE virtualizes the PS by hosting multiple traditional virtual machines, and manages the programmable logic by treating the accelerators as virtual devices that can be shared among the VMs.

Departing from the usage model of μ -VF, in Ker-ONE, users cannot load arbitrary hardware logic, but must instead select an accelerator from a predefined catalog, whose bitstreams are already known to the system and designed to support preemption. The accelerators are exposed to each VM as peripherals mapped in memory at predefined addresses. When an application inside a VM tries to use an accelerator that is not currently loaded on the PL, it writes to a control register mapped as read-only that is intercepted by the hypervisor and interpreted

as an implicit request for the hardware resource. At this point, the hypervisor consults an internal table, which lists the available pre-compiled bitstreams and their compatibility with the different reconfigurable regions. If it finds a solution, the hypervisor loads the requested bitstream on the PL.

μ -VF, on the other hand, is designed to guarantee each tenant a private region (Lake) where they can instantiate any custom design with a standard interface.

Similarly to FOS, Ker-ONE focuses on the management of computational accelerators and does not address the virtualization of GPIO.

CHAPTER

6

Conclusions

In this work, we addressed the fundamental mismatch between modern SoC-FPGA capabilities and the demanding requirements of edge computing applications. While existing FPGA virtualization frameworks excel in datacenters, they fall short at the edge due to their dependence on external orchestration and neglect of GPIO virtualization.

We presented μ -VF, a lightweight, fully autonomous virtualization framework that operates entirely on-device. At its core, μ -VF introduces the *virtual FPGA* (vFPGA) abstraction, granting each tenant a complete execution environment with containerized software on the PS and dedicated PL regions (*Lakes*). Unlike prior approaches, μ -VF's hypervisor runs directly on the embedded FPGA, enabling persistent applications with tight hardware-software coupling.

Our hardware-based GPIO virtualization layer, Abstract Tenant-Led Access to Signals (ATLAS), dynamically maps virtual pins to physical GPIOs, achieving: (i) abstraction from board-specific pin layouts; (ii) isolated peripheral access in multi-tenant environments; and (iii) 20ns reconfiguration latency at 100MHz. Once configured, ATLAS establishes purely combinatorial paths between virtual and physical pins where tenant logic drives GPIOs directly without PS involvement.

Performance evaluation on the AMD ZCU102 validates our design. The virtualization layer consumes less than 10% of FPGA resources, leaving 85% for tenant applications. Runtime overheads remain minimal: MMIO latency increases by only 2.93% (single-tenant) to 6.5% (four concurrent tenants), while memory throughput overhead stays below 1.8%, achieving 10,616 MB/s aggregate, 17.1% higher than previous approaches.

Future work will focus on: (i) OpenStack [21] integration for IaaS deployment; (ii) Advanced GPIO scheduling with quality of service (QoS) guarantees for time-multiplexed I/O sharing; (iii) Hybrid execution models supporting both persistent vFPGAs and dynamic tasks; and (iv) Overlay architectures [4] for bitstream portability across heterogeneous FPGAs.

By demonstrating comprehensive FPGA virtualization within embedded constraints, μ -VF transforms edge FPGAs from single-purpose accelerators into flexible, multi-tenant computing infrastructure.

To facilitate its adoption and encourage further development, the framework is publicly available as an open-source project [36].

References

- [1] Hedi Abdelkrim, Slim Ben Othman, and Slim Ben Saoud. “Reconfigurable SoC FPGA based: Overview and trends”. In: *2017 International Conference on Advanced Systems and Electric Technologies (IC-ASET)*. IEEE. 2017, pp. 378–383.
- [2] AMD Xilinx. *DFX Decoupler v1.0 LogiCORE IP Product Guide*. <https://docs.amd.com/r/en-US/pg375-dfx-decoupler>. PG375. 2023. URL: <https://docs.amd.com/r/en-US/pg375-dfx-decoupler>.
- [3] Christophe Bobda et al. “The Future of FPGA Acceleration in Datacenters and the Cloud”. In: *ACM Trans. Reconfigurable Technol. Syst.* 15.3 (Feb. 2022). ISSN: 1936-7406. DOI: 10.1145/3506713. URL: <https://doi.org/10.1145/3506713>.
- [4] Alexander Brant and Guy GF Lemieux. “ZUMA: An open FPGA overlay architecture”. In: *2012 IEEE 20th international symposium on field-programmable custom computing machines*. IEEE. 2012, pp. 93–96.
- [5] Ignacio Bravo-Muñoz, Alfredo Gardel-Vicente, and José Luis Lázaro-Galilea. *New Applications and Architectures Based on FPGA/SoC*. 2020.
- [6] Stuart Byma et al. “Fpgas in the cloud: Booting virtualized hardware accelerators with openstack”. In: *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE. 2014, pp. 109–116.
- [7] Jason Cong et al. “FPGA HLS today: successes, challenges, and opportunities”. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 15.4 (2022), pp. 1–42.

- [8] Salvatore Distefano, Giovanni Merlino, and Antonio Puliafito. “Sensing and Actuation as a Service: A New Development for Clouds”. In: *2012 IEEE 11th International Symposium on Network Computing and Applications*. 2012, pp. 272–275. DOI: 10.1109/NCA.2012.38.
- [9] William Fornaciari and Vincenzo Piuri. “Virtual FPGAs: Some steps behind the physical barriers”. In: *International Parallel Processing Symposium*. Springer. 1998, pp. 7–12.
- [10] Muhammed Kawser Ahmed et al. “Multi-Tenant Cloud FPGA: A Survey on Security, Trust, and Privacy”. In: *ACM Transactions on Reconfigurable Technology and Systems* 18.2 (2025), pp. 1–44.
- [11] Ahmed Khawaja et al. “Sharing, protection, and compatibility for reconfigurable fabric with {AmorphOS}”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 107–127.
- [12] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. “Do {OS} abstractions make sense on {FPGAs}?” In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 991–1010.
- [13] Roman L Lysecky et al. “Firm-core virtual FPGA for just-in-time FPGA compilation”. In: *FPGA*. 2005, p. 271.
- [14] Jiacheng Ma et al. “A hypervisor for shared-memory FPGA platforms”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 827–844.
- [15] Joel Mbongue et al. “FPGAVirt: A novel virtualization framework for FPGAs in the cloud”. In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE. 2018, pp. 862–865.
- [16] Paolo Meloni et al. “NEURAghe: Exploiting CPU-FPGA synergies for efficient and flexible CNN inference acceleration on Zynq SoCs”. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETs)* 11.3 (2018), pp. 1–24.

- [17] Panagiotis Miliadis et al. “Architectural support for sharing, isolating and virtualizing FPGA resources”. In: *ACM Transactions on Architecture and Code Optimization* 21.2 (2024), pp. 1–26.
- [18] Mohamad Najem et al. “Extended overlay architectures for heterogeneous FPGA cluster management”. In: *Journal of Systems Architecture* 78 (2017), pp. 1–14.
- [19] Gerald J Popek and Robert P Goldberg. “Formal requirements for virtualizable third generation architectures”. In: *Communications of the ACM* 17.7 (1974), pp. 412–421.
- [20] Ling Qian et al. “Cloud computing: An overview”. In: *IEEE international conference on cloud computing*. Springer. 2009, pp. 626–631.
- [21] Tiago Rosado and Jorge Bernardino. “An overview of openstack architecture”. In: *Proceedings of the 18th international database engineering & applications symposium*. 2014, pp. 366–367.
- [22] Nicolas Serrano, Gorka Gallardo, and Josune Hernantes. “Infrastructure as a service and cloud technologies”. In: *IEEE software* 32.2 (2015), pp. 30–36.
- [23] Douglas J Smith. *HDL Chip Design: A practical guide for designing, synthesizing and simulating ASICs and FPGAs using VHDL or Verilog*. Doone publications, 1998.
- [24] Hayden Kwok-Hay So and Cheng Liu. “FPGA overlays”. In: *FPGAs for software programmers*. Springer, 2016, pp. 285–305.
- [25] Mattia Tibaldi and Christian Pilato. “A survey of FPGA optimization methods for data center energy efficiency”. In: *IEEE Transactions on Sustainable Computing* 8.3 (2023), pp. 343–362.
- [26] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. “A survey on FPGA virtualization”. In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2018, pp. 131–1317.
- [27] Anuj Vaishnav et al. “FOS: A modular FPGA operating system for dynamic workloads”. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 13.4 (2020), pp. 1–28.

- [28] Tian Xia et al. “Ker-ONE: A new hypervisor managing FPGA reconfigurable accelerators”. In: *Journal of Systems Architecture* 98 (2019), pp. 453–467.
- [29] Sadegh Yazdanshenas and Vaughn Betz. “Quantifying and mitigating the costs of FPGA virtualization”. In: *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2017, pp. 1–7.
- [30] Yue Zha and Jing Li. “Hetero-ViTAL: A virtualization stack for heterogeneous FPGA clusters”. In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2021, pp. 470–483.
- [31] Yue Zha and Jing Li. “Virtualizing FPGAs in the cloud”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 845–858.
- [32] Jiansong Zhang et al. “The Feniks FPGA operating system for cloud computing”. In: *Proceedings of the 8th Asia-Pacific Workshop on Systems*. 2017, pp. 1–7.
- [33] Qi Zhang and Nirwan Ansari. “Virtualization”. In: *Encyclopedia of Cloud Computing*. Springer, 2016, pp. 1155–1161.

Sitography

- [34] *AMD Introduction to Dynamic Function eXchange*. 2025. URL: <https://docs.amd.com/r/en-US/ug909-vivado-partial-reconfiguration/Introduction-to-Dynamic-Function-eXchange>.
- [35] *AXI SmartConnect*, url = <https://docs.amd.com/r/en-US/pg247-smartconnec>,
- [36] Vincenzo Alessio Bucaria. *μ-VF GitHub Repository*. 2025. URL: <https://github.com/vincenzobucaria/u-VF-Enabling-Virtualization-of-Embedded-FPGAs>.
- [37] *DFX Decoupler*, url = <https://docs.amd.com/r/en-US/pg375-dfx-decoupler>,
- [38] *PYNQ*, url = <https://github.com/Xilinx/Pynq>,

Ringraziamenti

Desidero esprimere un sincero ringraziamento ai Prof. Longo e Merlino, per la loro preziosa guida, la competenza e le idee stimolanti che hanno accompagnato il mio percorso di studio e di ricerca.

Ringrazio la mia preziosa famiglia, che ha sempre fatto di tutto per sostenermi e supportarmi. Il loro affetto incondizionato è stato fondamentale durante questi anni di studio e lo sarà sempre. A mia madre, che da piccolo mi ricordava di dare un po' di tregua allo schermo; a mio padre, che sempre da piccolo mi metteva in guardia dai virus notturni del computer; e a mio fratello, che voleva (e vuole) solo andare a raidare su Rust. Ai miei nonni, Carmela e Vincenzo, che pensavano fossi già dottore in ingegneria solo perché "riparo" i loro televisori cambiando la sorgente.

Ovviamente non posso non ringraziare il mio amore, Vanessa, la mia forza e la mia ispirazione costante. Per lei io studio "i cavi", ma va bene così. In fondo, la laurea mi permette di fare un salto di livello: prima mettevo le lampadine, adesso studio i cavi. La sua presenza nella mia vita è un dono prezioso, e sono davvero fortunato ad averla al mio fianco. .

Ci tengo a ringraziare i miei cari cugini, gli zii, i miei splendidi amici, che mi hanno accompagnato con affetto e allegria in questi anni.

Infine, ringrazio me stesso!