μ-VF: Enabling Virtualization of Embedded FPGAs

VINCENZO ALESSIO BUCARIA, University of Messina, Italy
FRANCESCO LONGO, University of Messina, Italy
GIOVANNI MERLINO, University of Messina, Italy
FRANCESCO RESTUCCIA, Northeastern University, United States of America

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 paper 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 20.58% higher than previous frameworks. Hardware-based GPIO remapping completes in 20 nanoseconds.

CCS Concepts: • **Hardware → Reconfigurable logic and FPGAs**; • **Computer systems organization → Embedded and cyber-physical systems**; *Sensors and actuators*; *System on a chip*; Reliability; • **General and reference** → *Design*; *Experimentation*; *Evaluation*; • **Networks** → *Cloud computing*; • **Security and privacy → Virtualization and security**.

Additional Key Words and Phrases: FPGA virtualization, multi-tenancy, edge computing, resource isolation, IoT, IaaS, SoC-FPGA, I/O virtualization, hypervisor, containers

ACM Reference Format:
Vincenzo Alessio Bucaria, Francesco Longo, Giovanni Merlino, and Francesco Restuccia. 2025. μ-VF: Enabling Virtualization of Embedded FPGAs. *Proc. ACM Meas. Anal. Comput. Syst.* 9, 3, Article 66 (December 2025), 26 pages. https://doi.org/10.1145/3771581

Authors' Contact Information: Vincenzo Alessio Bucaria, vincenzo.bucaria@unime.it, University of Messina, Messina, Italy; Francesco Longo, francesco.longo@unime.it, University of Messina, Messina, Italy; Giovanni Merlino, giovanni.merlino@unime.it, University of Messina, Messina, Italy; Francesco Restuccia, f.restuccia@northeastern.edu, Northeastern University, Boston, United States of America.

ACM 2476-1249/2025/12-ART66
https://doi.org/10.1145/3771581

1 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. Field-Programmable Gate Arrays (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 Input/Output (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 [3, 9] and enabling multiple applications to efficiently share hardware resources.

Existing work on FPGA virtualization [14, 16, 28] 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 [27][11]. These approaches do not extend naturally to SoC-class FPGAs, where Processing System (PS) and Programmable Logic (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 [22], depend on high-power (e.g., x64-hosted) orchestration over PCIe [24], and exceed the form factor and thermal envelope of mobile or battery-powered platforms.

Existing frameworks for embedded FPGAs [25, 26] 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 [1, 5, 15].

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 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

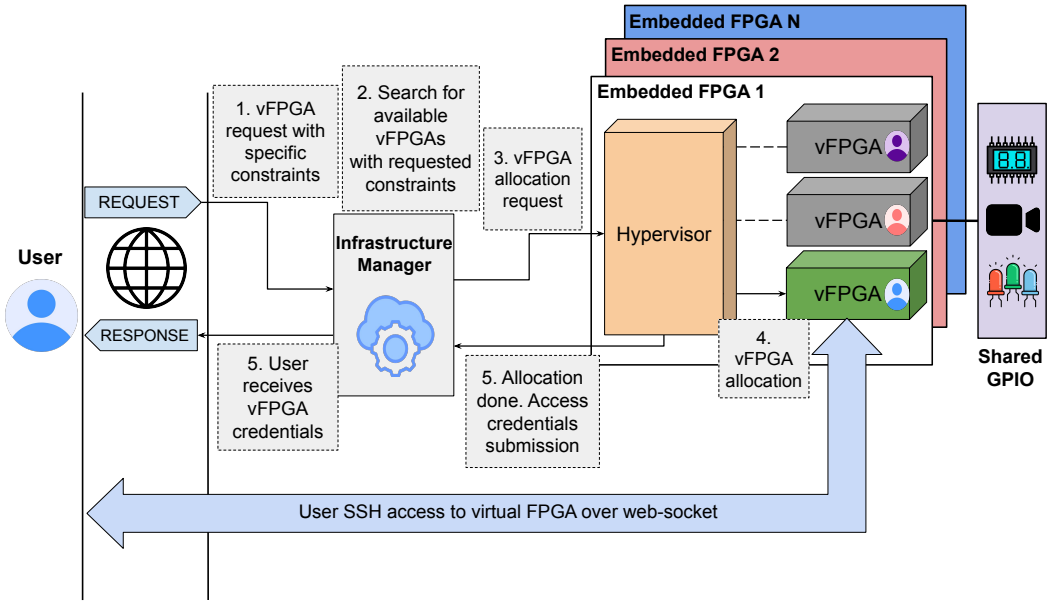

Fig. 1. The μ-VF framework implements a Distributed Infrastructure-as-a-Service model where each embedded FPGA exposes multiple virtual FPGA (vFPGA) that comprise both PS resources (containerized execution environments) and PL resources (isolated hardware regions). While each device is designed to operate autonomously, an optional central infrastructure manager can be employed to orchestrate vFPGA allocation across a distributed fleet. Users connect directly to their assigned vFPGA and obtain dedicated PS/PL resources.

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 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 [25], which, on the same ZCU102 platform, were limited to around 50% resource allocation due to virtualization overhead.

The key contributions of this paper can be summarized as follows:

- a fully on-device virtualization stack for embedded FPGAs, enabling secure multi-tenant execution with minimal resource overhead;
- dynamic General-Purpose Input/Output (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) Memory-Mapped I/O (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.

This paper is organized as follows. Section 2 provides background on SoC-FPGA architectures and edge computing challenges. Section 3 details the architecture of μ-VF. Section 4 evaluates the performance of the framework. Section 5 provides an in-depth analysis of related work, while Section 6 concludes the paper.

2 Background and Preliminary Concepts

This section provides the necessary background on SoC-FPGA architectures and edge computing service models that motivate our work. We first introduce the hardware platform targeted by μ-VF, then explain how Partial Reconfiguration (PR) enables multi-tenancy, discuss the service abstractions needed for edge deployments, and finally outline the unique challenges of virtualizing FPGAs in resource-constrained environments.

2.1 SoC-FPGA Architecture

FPGAs are chips containing reconfigurable logic blocks that can be programmed to create custom hardware circuits, offering higher performance and energy efficiency than traditional processors for specific tasks. Modern System-on-Chip FPGAs (SoC-FPGAs) integrate both traditional processors (PS) and programmable logic (PL) on a single chip. For instance, the AMD Zynq UltraScale+ ZCU102 combines ARM cores running Linux with reconfigurable fabric, where the PS handles software control while the PL implements hardware accelerators. Unlike datacenter FPGAs that depend on external servers, SoC-FPGAs are self-contained systems ideal for edge deployments where infrastructure is limited.

2.2 Partial Reconfiguration

PR enables modifying portions of the FPGA fabric while the rest continues operating. This technology is fundamental for multi-tenancy, allowing independent loading and unloading of tenant accelerators without disrupting co-located workloads. Modern FPGAs support PR through dedicated configuration ports accessible from the PS.

2.3 Edge Computing Service Models

Edge computing extends cloud paradigms to distributed resources at the network periphery. Two service models are particularly relevant for edge FPGA deployments:

The Infrastructure as-a-Service (Iaas) [20] paradigm virtualizes compute resources, allowing multiple tenants to deploy workloads on shared infrastructure while maintaining isolation. In the context of FPGAs, this means abstracting heterogeneous hardware behind a unified interface, enabling users to deploy accelerators without managing board-specific details.

Sensing and Actuation-as-a-Service (SAaaS) [8] extends virtualization to physical devices, treating sensors and actuators as requestable resources. This model is crucial for edge scenarios where applications need not just computation but also interaction with the physical world — for instance, environmental monitoring requiring both data processing and access to distributed temperature sensors.

2.4 FPGA Virtualization Challenges

Virtualizing embedded FPGAs for edge deployments presents unique challenges:

- **Resource Constraints**: Edge FPGAs have 10-100× fewer logic resources than datacenter accelerators, requiring minimal virtualization overhead.

- **Autonomous Operation**: Without external hosts, all virtualization logic must run on-device within tight power budgets.
- **Direct I/O Access**: Edge applications require deterministic and low-latency access to sensors and actuators through GPIO pins in order to interact with the physical world.
- **Heterogeneous Platforms**: Different FPGA boards have varying resources and pin layouts, necessitating abstraction layers for portability.

3 μ-VF Architecture

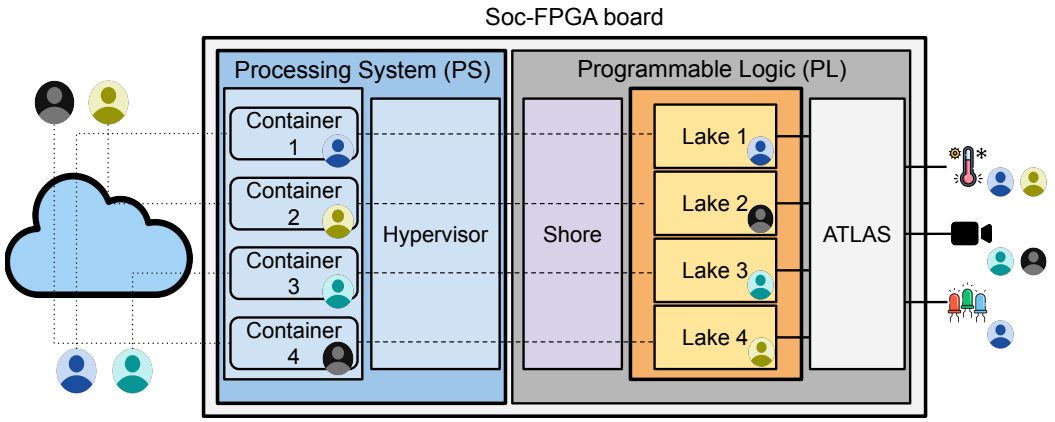

Fig. 2. Overview of the μ-VF architecture. Each tenant operates within an isolated software container running on the Processing System (PS), with access to a private Lake in the programmable logic (PL). The Shore manages access to shared resources such as DDR memory and provides the communication interface 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.

Figure 2 illustrates the overall architecture of μ-VF. Each μ-VF-enabled FPGA exposes a unified resource abstraction that spans both the programmable logic (PL) and the on-chip processing system (PS).

On the PL side, the fabric is partitioned into multiple reconfigurable regions called *Lakes*, isolated subsets of gates within the FPGA's sea of resources, each dynamically assigned to a tenant for hardware accelerator deployment. These Lakes are physically and logically isolated, and interact with a shared static *Shore* that mediates access to common services, including memory, reconfiguration, and processing system interaction.

ATLAS (Abstract Tenant-Led Access to Signals), the GPIO virtualization layer implemented entirely within the PL, enables direct, low-latency connection between tenant logic and physical I/O devices, without routing through the PS.

In parallel, the PS hosts a set of lightweight software containers, one per tenant, which provide isolated execution environments for control applications. A minimal on-device hypervisor manages these containers, handles resource arbitration, and coordinates partial reconfiguration. All tenant interactions with hardware resources, including memory buffers, GPIO mappings, and configuration requests, are mediated through a secure API exposed by the hypervisor.

In essence, this architecture delivers true on-device virtualization by providing each tenant with a complete virtual FPGA (vFPGA). This vFPGA acts as the tenant's private resources *slice*,

encompassing both hardware and software, which provides a sense of full ownership while the system enforces strict isolation and manages resource sharing.

The remainder of this section delves into the main architectural components of μ-VF, covering in turn: (i) the Hardware Stack, comprising Lakes for tenant isolation, ATLAS for GPIO virtualization, and the Shore infrastructure for managing shared resources; and (ii) the Software Stack, including the containerized execution model on the PS and the embedded hypervisor responsible for runtime coordination.

3.1 Hardware Stack

The hardware stack design balances two critical requirements for embedded FPGAs: minimal resource footprint and performance. This is achieved through a hybrid approach where time-critical operations (e.g., GPIO routing, memory access control) are implemented in hardware, while management functions (e.g., policy decisions, resource allocation) are delegated to the PS hypervisor. The hardware stack comprises two categories of components:

- Static components (Shore and ATLAS) are synthesized and programmed onto the FPGA once at system initialization and persist throughout the device's operation, providing the foundational virtualization services.
- Dynamic components (Lakes) can be allocated, programmed, and de-allocated independently at runtime through partial reconfiguration (PR). When a tenant requests to deploy or update their accelerator, only that tenant's specific Lake is loaded through the FPGA's configuration port, leaving other Lakes and the static infrastructure undisturbed.

The following subsections detail each component's implementation.

3.1.1 Lakes: Isolated Regions with Standardized Interfaces. Each tenant is assigned a Lake, corresponding to a private partially reconfigurable region within the FPGA fabric. Within this region, users are free to implement application-specific hardware accelerators using either traditional hardware description languages (HDLs) [21] or high-level synthesis (HLS) tools [7], provided their design conforms to a standardized top-level interface. As shown in Figure 3, each tenant's hardware design must be encapsulated within a Lake wrapper that exposes a fixed interface to the static infrastructure. This wrapper abstracts the complexity of the virtualization layer while providing standard connection points for all required services. Once synthesized, these designs can be dynamically deployed at runtime through partial reconfiguration, allowing tenants to load, update, or swap their accelerators on-demand without affecting other running workloads. The Lake interface is explicitly designed to integrate with the surrounding shell infrastructure and reflects the typical requirements of embedded-class systems. Specifically, each partition may require access to: (i) coordination with software processes running on the PS, (ii) shared memory subsystems (e.g., DDR), and (iii) external peripherals such as sensors, actuators or off-chip I/O.

To support these requirements, the interface exposes three classes of communication primitives:

- a control channel, implemented as a 32-bit AXI-lite bus, enabling memory-mapped access from the PS to configuration registers within the user logic. The AXI-Lite protocol is chosen for its simplicity and low resource footprint, as control operations typically involve sporadic register read/writes that do not require high bandwidth.
- a data channel, implemented as a 128-bit AXI-Full bus, which supports high-throughput transactions initiated by the role toward shared memory resources such as DDR. The wider bus width and full AXI protocol accommodate bandwidth-intensive applications, supporting burst transfers to match the memory subsystem capabilities.
- a set of tri-state GPIO signals, intended for interaction with external peripherals through ATLAS. These signals follow the standard tri-state convention where each logical GPIO

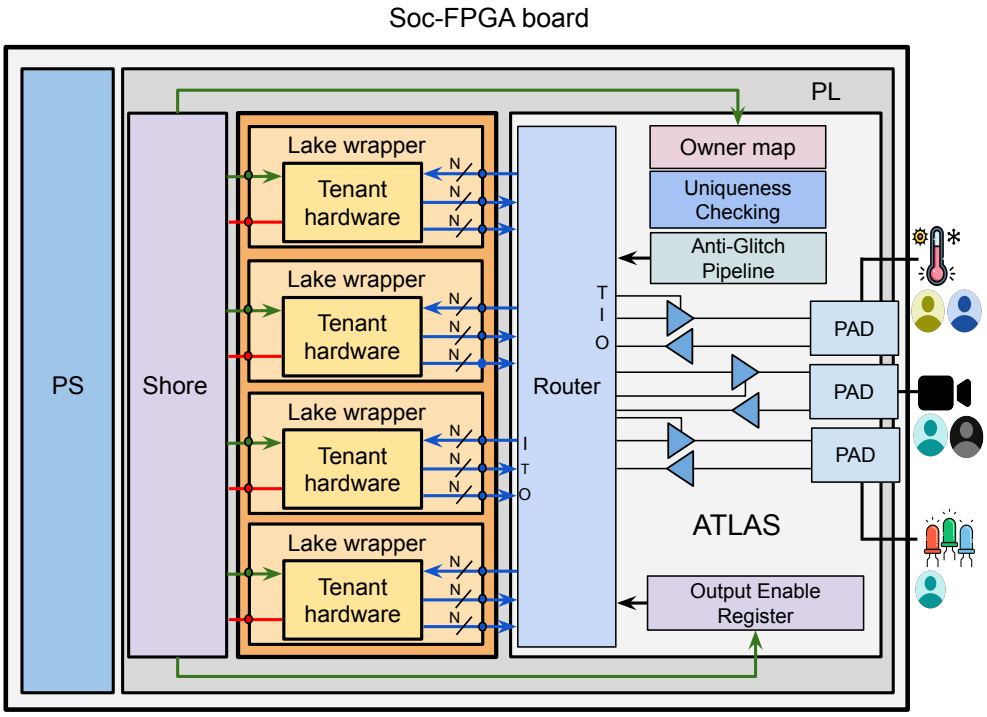

Fig. 3. ATLAS GPIO virtualization architecture in μ-VF. Tenant hardware within Lake wrappers exposes N tri-state virtual pins (I/O/T signals) that ATLAS routes to physical pads via a configurable router. The Owner Map and Anti-Glitch Pipeline ensure safe dynamic remapping, while the Output Enable Register 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.

consists of three wires: data input (I), data output (O), and output enable (T), allowing bidirectional communication with external devices.

Using standard interfaces enforces a write-once, deploy-anywhere development paradigm. While bitstreams are not portable and require synthesis targeting the specific board, the source design remains portable across different FPGAs that conform to the same interface specification. This architectural decoupling enhances the reusability of tenant designs and facilitates integration into diverse deployment targets.

While standardized bus interfaces like AXI are industry norms widely adopted across 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 [25] [26]. The μ-VF approach tackles this gap: instead of binding hardware logic to board-specific physical pins, each Lake exposes virtual pins as logical placeholders. These virtual pins are dynamically mapped to physical GPIOs at runtime through ATLAS, our hardware-based virtualization layer. This design offers several important advantages. First, it abstracts away board-level pin assignments, allowing developers to implement hardware without concern for the specific physical I/O layout of the underlying platform. As a result, user designs can be seamlessly migrated across different Lakes or even across heterogeneous FPGAs without requiring manual modifications to pin-level logic. Furthermore, the runtime remapping capability

supports fault recovery through peripheral reassignment and enables dynamic resource allocation based on changing workload demands. Most critically, it provides secure, isolated peripheral access in multi-tenant environments: a capability absent in existing FPGA virtualization frameworks.

3.1.2 ATLAS: Autonomous Tenant-Led Access to Signals. The realization of these benefits requires a hardware mechanism that can dynamically route 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 wherein I/O lines must be electrically and directly accessible from user hardware-logic, with minimal latency and without software mediation.

As illustrated in Figure 3, ATLAS implements a crossbar-based routing architecture with integrated safety mechanisms:

- Dynamic Routing Matrix: at its core, ATLAS employs a configurable crossbar that maps virtual pins to physical PADs based on the 'owner_map' configuration register, which is dynamically updated by the software hypervisor at runtime. Each Lake exposes a fixed number of virtual pins, and ATLAS manages N total virtual pins (the sum across all Lakes). The owner_map uses values 0 to N, where 0 indicates unmapped and values 1 through N map to virtual pins 0 through 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 combinational path, preserving the same electrical characteristics as in a non-virtualized system.
- Uniqueness Validation: 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 across all pins: if multiple physical pins are configured to connect to the same virtual pin, only the first valid assignment is honored while others are safely disconnected.
- Dual-Stage Output Control: output driving follows a two-level permission model. First, each Lake controls its virtual pins through standard tri-state signals, as already discussed. Second, the hypervisor maintains per-physical-pin output enable flags. Thus, a physical pin is driven only when both conditions are met. This dual control prevents unauthorized peripheral access even if a malicious Lake attempts to drive outputs.
- Anti-Glitch Pipeline: the architecture inherently prevents glitches during reconfiguration through registered state updates. When the hypervisor updates the owner_map, the new configuration is first validated (uniqueness check), then atomically applied on the next clock edge.

By maintaining purely combinatorial paths from virtual pins to physical PADs, ATLAS preserves the electrical characteristics and timing of direct physical connections. This hardware-based approach to GPIO virtualization represents a key enabler for edge-native multi-tenancy, addressing a critical gap in existing FPGA virtualization frameworks that focus solely on computer and memory resources while neglecting the diverse I/O requirements of embedded applications.

3.1.3 Shore Infrastructure and Isolation Mechanisms. As illustrated in Figure 4, the Shore implements essential infrastructure for secure resource sharing and Lake isolation. The architecture employs a modular design where each Lake interfaces with dedicated Shore components to ensure both performance and security.

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 its associated Lake from the Shore infrastructure, preventing any signals from propagating in

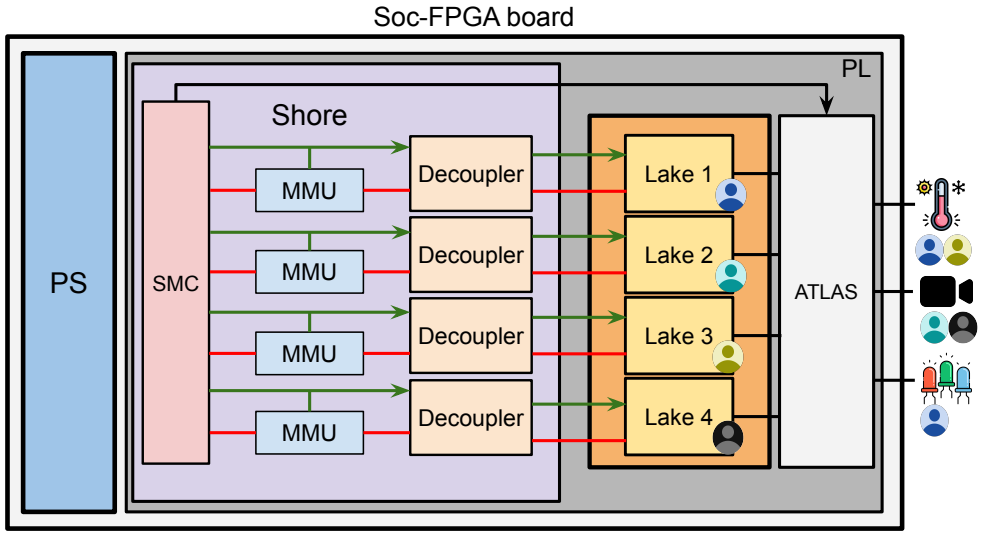

Fig. 4. μ-VF Shore architecture. The Shore layer mediates between PS and PL, with per-Lake Memory Management Units (MMUs) that allow the hypervisor to configure memory access permissions for each tenant's hardware. Decouplers provide electrical isolation during reconfiguration. **Legend.** Red: AXI-Full bus; Green: AXI-Lite bus; Black: GPIO signals.

either direction. This isolation mechanism serves a dual purpose: it ensures 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 might attempt to disrupt system operation.

Memory Management Units (MMUs) manage memory access from user designs. The controllers are implemented directly in hardware to minimize performance overhead and latency, ensuring efficient communication with the DDR subsystem, while also being 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 enforces strict access isolation, 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 buffer allocation from its software container on the PS, the hypervisor allocates the memory accordingly and configures the hardware controller to grant access exclusively to the relevant physical address range.

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

3.2 Processing System Virtualization and Software Stack

With edge-class deployments in mind, μ-VF adopts a fully on-device execution model in which each tenant's software stack runs directly on the same FPGA platform as their hardware logic. This design choice is driven by fundamental constraints of edge environments:

First, executing control software on a remote host introduces unavoidable latency in hardware-software interactions. Consider a typical scenario where the control software needs to respond

to a sensor event: the FPGA accelerator detects the event and must notify the remote host over the network, the host processes the event and decides on an action, then sends commands back to the FPGA. Even with low-latency networks, this round-trip communication adds milliseconds of delay. For edge applications requiring microsecond-scale response times, such as motor control in robotics or real-time anomaly detection, these latencies are prohibitive.

Second, power budgets at the edge are severely limited. External host servers would add significant power overhead (exceeding the FPGA itself) making battery-powered deployments infeasible.

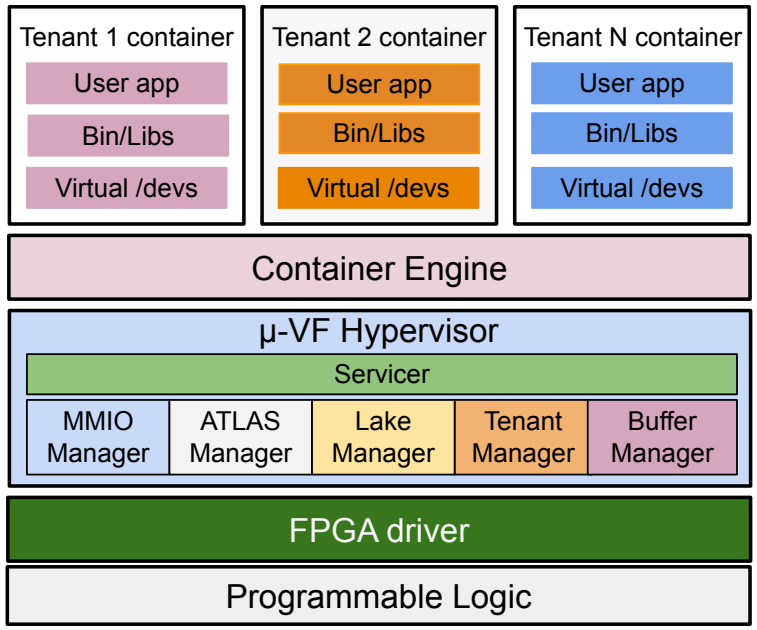

Fig. 5. Software stack of µ-VF. Each tenant runs inside a containerized environment, supported by a container engine. The µ-VF hypervisor manages memory-mapped access, resource allocation, and partial reconfiguration through modular, multi-threaded services. A dedicated FPGA driver interfaces with the underlying programmable logic.

As illustrated in Figure 5, the PS is partitioned into lightweight, isolated containers, each hosting a tenant's application code. This containerized architecture provides strong isolation between tenants while offering each of them a private, self-contained execution environment. Within this environment, application code can interact tightly and with minimal latency with the user's hardware logic in the fabric, leveraging shared memory buffers and fast communication pathways that are only possible with on-device integration. Beyond isolating tenants from one another, a containerized architecture also decouples each tenant from direct access to the underlying FPGA resources. This abstraction enhances the overall security and robustness of the system, as tenants are only allowed to interact with shared resources through the supervision of the on-device hypervisor.

3.2.1 On-device Hypervisor. The supervisory layer is implemented as a modular service architecture, where specialized components handle 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. Upon receiving a request, it dispatches the operation to the appropriate internal component.

The *Tenant Manager* maintains the system state for each tenant, including authentication, authorization, and the tracking of allocated resources. It enforces policy constraints, such as the maximum number of buffers, the permissible hardware designs, the accessible partial regions in the programmable logic (PL), the set of assignable GPIOs, and the physical slave interfaces available to the tenant. Enforcement of these constraints is delegated to other specialized subsystems.

The *Lake Manager* handles the secure deployment of user hardware designs into 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 requests for memory allocation, 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* handles interactions with the fabric-level I/O virtualization layer. It performs the dynamic mapping of each role's virtual GPIO pins to physical board pins, based on system policy and runtime availability, allowing tenants to transparently access external peripherals without requiring static assignments or design changes.

The *MMIO Manager* is responsible for managing virtual memory-mapped 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.

3.2.2 Container-Based Tenant Execution. Figure 6 illustrates the software architecture and the interaction flow between tenant containers and the hardware resources. The software stack implements a capability-based access control system that dynamically manages hardware permissions based on Lake assignments. 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 data copying through intermediate buffers or hypervisor mediation.

For *MMIO access*, each container in the system is presented with N virtual MMIO devices (where N equals the number of Lakes), appearing as `/dev1` through `/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 through the kernel's memory mapping—no data copying or hypervisor intervention occurs in the data path. Access to these devices is strictly controlled through Linux file permissions (user/group ownership), ensuring that containers can only interact with their assigned Lake.

For high-throughput data transfers, μ-VF implements zero-copy shared memory buffers backed 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 (virtual) memory handle that allows both the software and hardware components to access the same physical memory without any intermediate copying.

When a tenant application needs to deploy an accelerator, it follows the sequence shown in Figure 6:

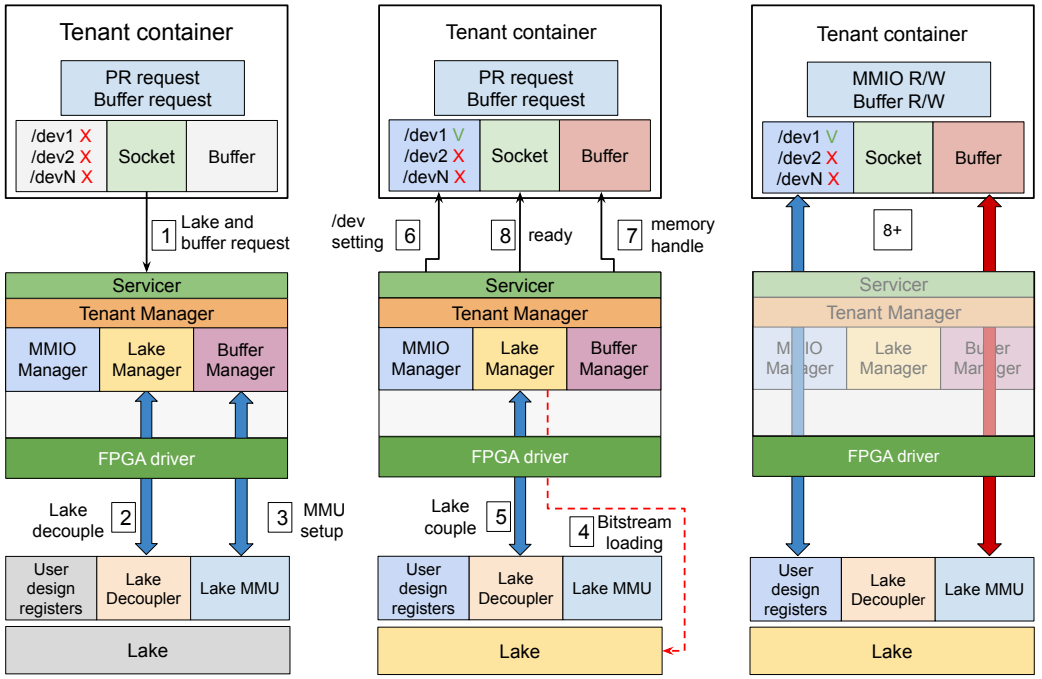

Fig. 6. Software architecture and resource allocation flow for multi-tenant FPGA virtualization. The diagram shows the interaction between tenant containers, hypervisor services (MMIO Manager, Lake Manager, Buffer Manager) and FPGA hardware resources, with numbered steps (1-8) indicating the complete 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 arrows show AXI Full connections.

(1) **Resource Request.** The tenant application initiates a PR (Partial Reconfiguration) request through socket-based communication to the hypervisor, specifying: (i) the bitstream to be loaded (previously registered with the hypervisor), (ii) required memory buffers and their sizes, and (iii) any specific Lake preferences.
(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 for reconfiguration.
(3) **MMU Configuration.** The hypervisor configures the MMU with 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 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 tenant use.
(6) **Permission Granting.** The hypervisor updates the file permissions of the corresponding /dev device, granting the requesting container exclusive access. For instance, if Lake 1 is assigned, the container gains read/write access to /dev1 through Linux user/group membership modifications.

(7) **Buffer Handle Transfer.** The hypervisor returns the handle of the contiguous memory buffer created for the tenant.
(8) **Resource Ready Notification.** The hypervisor notifies the client that resources are ready for use.

This design provides each tenant with the illusion of exclusive FPGA ownership while the hypervisor maintains system-wide 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.

3.3 Application Integration Model

The μ-VF architecture was specifically designed to be user-friendly. In particular, the software-stack APIs mirror the interface of the APIs provided by PYNQ. This design choice significantly lowers the entry barrier for developers already familiar with the PYNQ framework. By maintaining API consistency, users can immediately leverage their existing knowledge, and code examples developed for PYNQ, drastically reducing the learning curve associated with adopting the μ-VF platform.

3.4 Security and Isolation Analysis

While the architecture components described in previous sections incorporate security mechanisms, this section explicitly addresses critical security concerns that arise in multi-tenant FPGA environments.

3.4.1 Memory Access Violations. When a tenant attempts to access memory regions outside their allocation, multiple independent mechanisms block the operation. For MMIO accesses to control registers, each Lake is exposed through device-tree-segmented virtual devices with hardware-enforced address boundaries. Even if a tenant bypasses filesystem permissions, the virtual device remains physically constrained to its Lake's address range, making cross-Lake access impossible.

For hardware-initiated memory accesses, the MMUs in the Shore infrastructure validate every AXI transaction from Lake hardware against hypervisor-configured boundaries. Unauthorized access attempts are blocked at the hardware level before reaching the DDR controller. Additionally, the pynq_char_mapper driver's Tenant Table ensures tenants can only map CMA buffers explicitly allocated to them. Since tenants operate exclusively with virtual addresses, never seeing physical addresses, they cannot craft malicious memory requests to bypass these protections.

3.4.2 GPIO Control and Race Prevention. The ATLAS architecture prevents tenants from racing for GPIO control by design. Tenants interact only with virtual pins, while ATLAS maintains an OwnerMap ensuring each physical pin connects to at most one virtual pin. Only the hypervisor can modify this mapping through privileged AXI transactions. The planned scheduling system for ATLAS (future work) will remain entirely hypervisor-coordinated, eliminating any possibility of direct user manipulation of pin assignments.

3.4.3 Memory Bus Contention and Arbitration. The impact of concurrent resource access has been empirically characterized in our performance evaluation (Section 4). In our current implementation supporting up to four tenants, each Lake is assigned a dedicated High-Performance (HP) port to the memory subsystem, effectively isolating memory bandwidth between tenants.

However, this approach does not scale beyond four tenants due to the limited number of HP ports on the ZCU102 platform. For deployments requiring more than four concurrent tenants, multiple Lakes would need to share HP ports, introducing potential bandwidth contention. To address this limitation, our future work (Section 6) will implement a Quality-of-Service (QoS) layer that enables bandwidth reservation and priority-based arbitration when HP port sharing becomes necessary.

3.5 Portability

The μ-VF hardware architecture is fully portable to Xilinx SoC-FPGAs, as is the software stack, which was explicitly designed for PYNQ. However, while the architectural components can be instantiated simply by using a TCL script, both the floorplanning and constraints must be reorganized based on the target board. The operating clock frequency can also be selected according to the capabilities of the target FPGA, allowing designers to balance timing closure, power efficiency, and performance depending on deployment requirements.

4 Performance 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. It is important to clarify the scope of this evaluation. μ-VF is designed to virtualize the communication and access paths (MMIO, memory, GPIO), but it does not introduce any abstraction layer on the primitive compute resources (such as LUTs, DSPs, and BRAMs) themselves. Consequently, the execution performance of a tenant's core accelerator logic is not impacted. Therefore, this evaluation targets only the components affected by the virtualization stack: the resource overhead of the hardware, the latency and throughput of memory and I/O interactions, the responsiveness of dynamic GPIO reconfiguration, and the end-to-end responsiveness of resource allocation. We argue that these benchmarks are directly representative of the expected overhead for any real-world application. A full validation under specific application workloads, however, remains an important next step for future work, as will be discussed in Section 6.

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.

Table 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

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. 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. The design was implemented using the floorplan in Figure 8, with a programmable logic clock of 100 MHz and memory port assignments as detailed in Table 2.

Table 2. Per-tenant HP port assignment

Tenant	Assigned HP port
Tenant 1	HP 0
Tenant 2	HP 1
Tenant 3	HP 2
Tenant 4	HP 3

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 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.

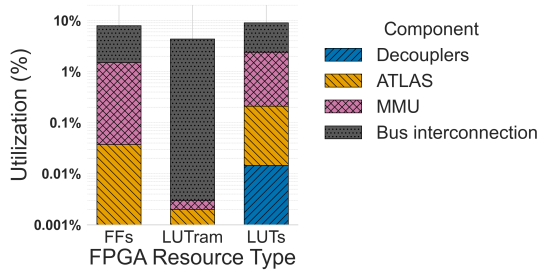

Fig. 7. Hardware stack resource usage breakdown after synthesis. Components include decouplers for electrical isolation, ATLAS for GPIO virtualization, MMUs for memory protection, and bus interconnection fabric. Total utilization remains below 10% for all resource types.

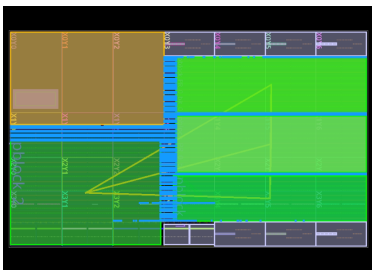

Fig. 8. Floorplanning view after synthesis and implementation. Lakes are represented by green areas, the processing system occupies the orange area, while the hardware stack (Shore and ATLAS) is shown in blue.

Figure 7 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 3 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 8, 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.

Table 3. 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%

Overall, the post-implementation allocation reserves approximately 85% of the total available resources for user workloads, as shown in table 4.

Table 4. Comparison of total FPGA resource allocation between µ-VF and FOS [25] on ZCU102.

Resource Type	µ-VF (4 Lakes)	Competing Approach (4 PRs)	Difference (pp)
CLB LUTs	85.47%	46.80%	+38.67
LUTram	85.56%	47.60%	+37.96
BRAM	86.07%	48.40%	+37.67
DSP	85.72%	53.20%	+32.52

This represents a significant improvement over prior approaches [25] 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 3.2.2, 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.

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 9 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

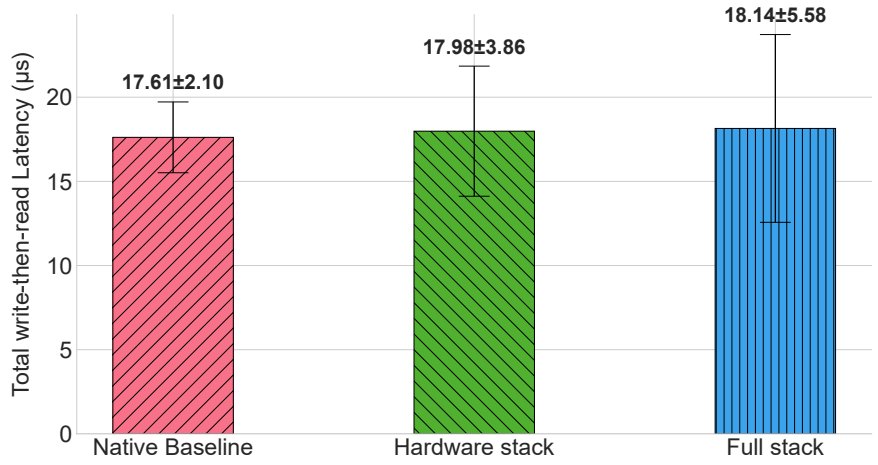

Fig. 9. MMIO access latency comparison between native (non-virtualized) execution and μ-VF with incremental virtualization layers in single-tenant mode. Measurements show: native baseline, hardware stack only (Shore + ATLAS), and full μ-VF stack (hardware + software virtualization).

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 μs of overhead, more than 4.4× the total latency of our zero-copy implementation.

Figure 10 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 μ-VF stack scales effectively across tenants, without introducing significant contention or delay in MMIO access.

Figure 11 shows the complete latency distribution histograms for MMIO operations under isolated and concurrent execution, illustrating their shape and sensitivity to OS noise. The distributions remain tightly bounded, with worst-case latencies of approximately 16 μs for write operations and 14 μs for read operations, even with four concurrent tenants.

The overlapping distributions of all four concurrent tenants confirm fair resource allocation, with no individual tenant experiencing disproportionate performance degradation. The narrow distribution width, and the abscence of a long tail, validate that μ-VF's zero-copy architecture is resilient to OS jitter and provides the deterministic latency bounds required for real-time FPGA applications.

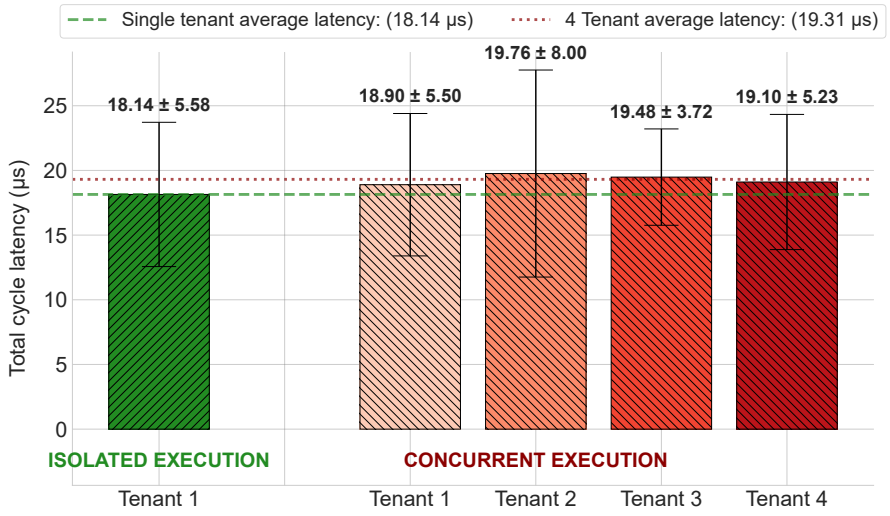

Fig. 10. MMIO access latency under isolated and concurrent execution with full µ-VF virtualization (hardware + software stack). The graph shows the average write-then-read cycle latency for a single tenant running alone (isolated execution) versus four tenants simultaneously performing read/write operations to their respective Lake registers (concurrent execution).

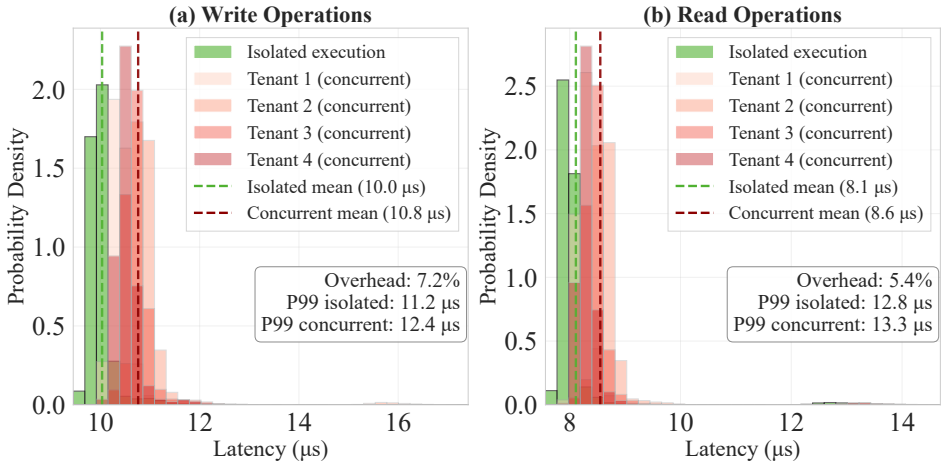

Fig. 11. MMIO latency distributions under isolated and concurrent four-tenant execution.

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 μ-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, Shore routing, and any buffering or interconnect overhead.

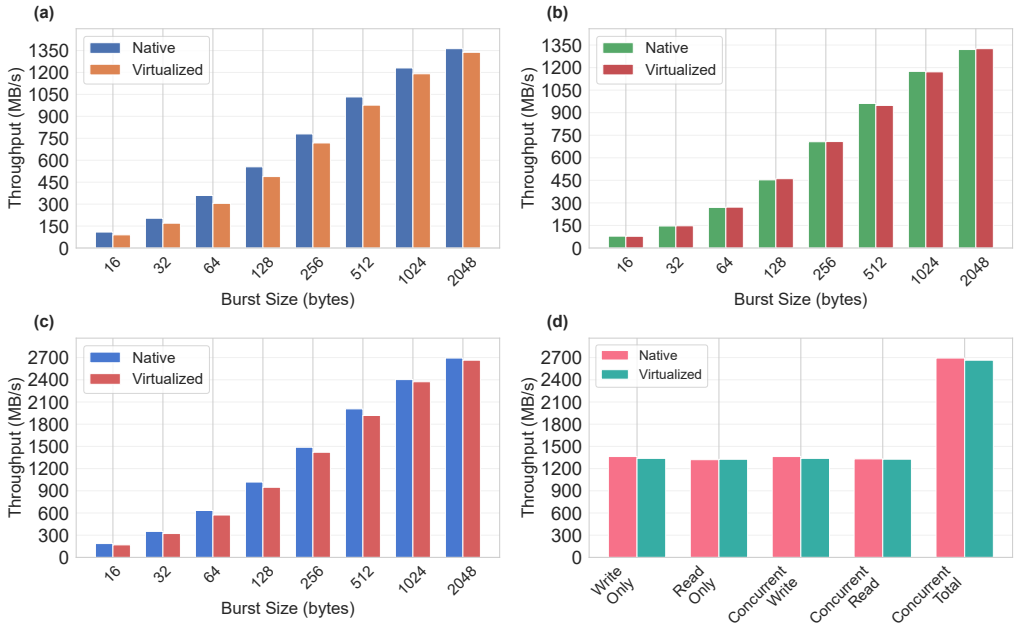

Fig. 12. Single-tenant memory throughput on HP0 port: native vs virtualized execution. (a) Write throughput, (b) Read throughput, (c) Concurrent R+W aggregate throughput across burst sizes from 16B to 2KB. (d) Summary at 2048B burst size. Each measurement point is the average of 100 samples. Error bars are omitted because the observed variability is negligible.

Our benchmarks compare the native baseline, the μ-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 12, 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.

In write-only tests, as shown in Figure 12(a), throughput dropped slightly from 1362.6 MB/s (native) to 1337.8 MB/s with the Shore, 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 12(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 12(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.

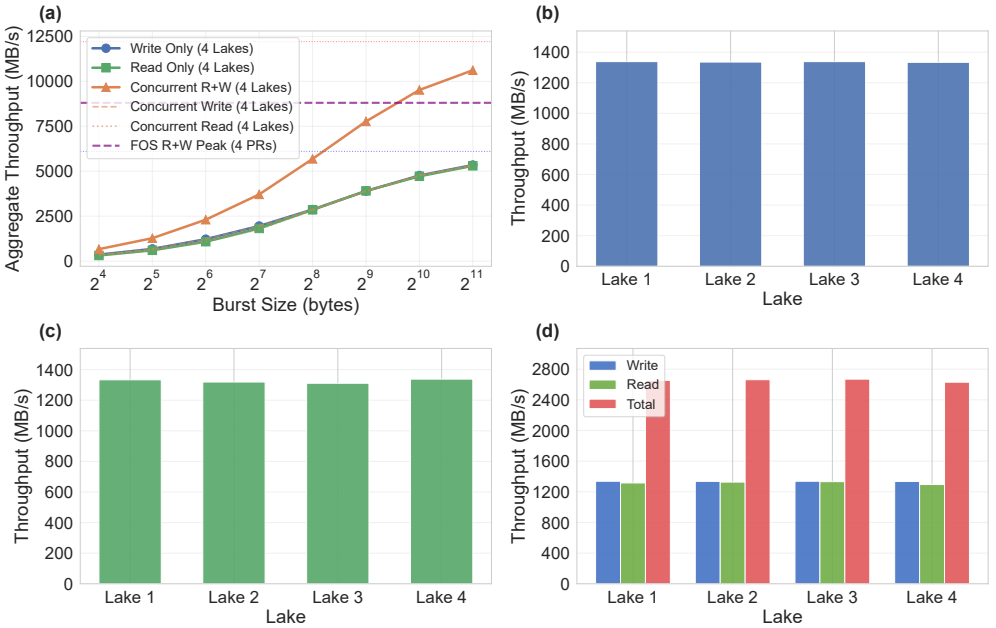

Fig. 13. Multi-tenant memory throughput with four concurrent Lakes, each assigned to a dedicated HP port. (a) Aggregate throughput vs. burst size compared to FOS four-tenant peak (8,804 MB/s for concurrent R+W on four PRs), achieving 10,616 MB/s total bandwidth. (b) Write-only throughput per Lake at 2048B. (c) Read-only throughput per Lake at 2048B. (d) Per-Lake breakdown of concurrent read/write throughput at 2048B showing individual bandwidth contributions. Each measurement point is the average of 100 samples. Error bars are omitted because the observed variability is negligible.

In Figure 13 we report throughput in the worst case scenario, i.e., the condition in which 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 13(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 13(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 13(a) shows that we achieve an aggregate throughput of 10,616 MB/s, representing a 20.58% improvement over previous approaches [25].

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 12, the measurement corresponding to access performed outside the containerized environment, reflecting the hardware-stack-only contribution relevant to the hypervisor, adds approximately 18 μs 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 μs at 100 MHz.

4.6 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.

The first set of results, shown in Figure 14, 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 15 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.

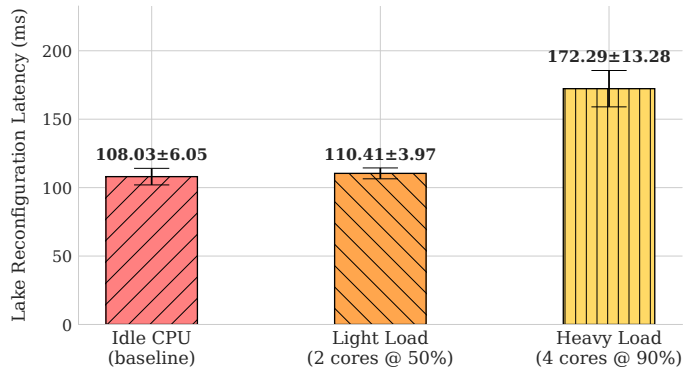

Fig. 14. Lake Reconfiguration Latency (Warm-Request) vs. CPU Load. The time measured includes the client's API request, bitstream allocation, and hypervisor response with metadata.

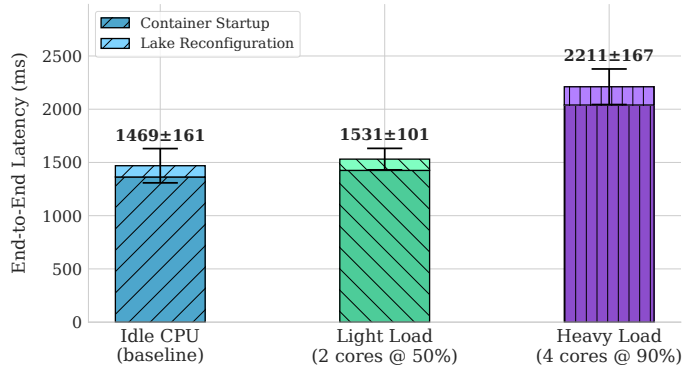

Fig. 15. End-to-End Latency (Cold-Request) Breakdown vs. CPU Load. Total time includes Container Startup and Lake Reconfiguration.

5 Related Work

One of the first approaches to FPGA virtualization in the Cloud is the one by Byma et al. [6], which integrates FPGAs into the Cloud by using OpenStack and partitioning just the programmable logic through partially reconfigurable regions. In such a work, while the network interface and DDR access are virtualized, GPIO virtualization is out of interest and the hypervisor runs on an external server. In the data center context, subsequent works focus on FPGA virtualization aiming at optimizing resource utilization and fragmentation due to allocations through statically-sized partial reconfigurations. AmorphOS [11] is an open-source operating system for PCI-E FPGA that introduces *Morphlets*, i.e., abstractions that encapsulate user logic and enable dynamic scaling based on available fabric resources. The system supports multi-tenancy through a hybrid scheduling approach that transitions between partial reconfiguration zones and global bitstreams to optimize resource utilization. A key innovation is the ability of user designs to dynamically resize their on-chip area at runtime. While implemented on Amazon F1 and Microsoft Catapult with PCIe and DRAM virtualization, GPIO and other peripheral interfaces remain unsupported.

Several other FPGA virtualization systems explore OS-like abstractions for multi-tenant execution. Coyote [12] provides comprehensive OS abstractions including virtual memory and processes.

ViTAL [28] and Hetero-ViTAL [29] focus on cluster-scale virtualization with compiler/runtime co-design. Optimus [13] addresses memory virtualization through page table slicing, while Feniks [30] ensures performance isolation between OS and application regions. However, all these systems target datacenter-class PCIe-attached FPGAs, assume abundant resources, and, critically, none address GPIO virtualization or standalone operation.

Moving to works focusing on FPGA-SoC platforms, as μ-VF does, it is important to contextualize them within the broader landscape of embedded virtualization. A significant body of research focuses on enhancing security on traditional, fixed-hardware embedded processors. These approaches often use lightweight hypervisors to partition a single application's software components into isolated VMs for threat containment, as demonstrated by Moratelli et al. [17] and Tiburski et al. [23]. While effective for software security on CPUs, these methods do not address the unique challenges of SoC-FPGAs: the secure management and sharing of reconfigurable hardware logic and physical I/O. In the SoC-FPGA context, some solutions like Virtio-FPGA [18] prioritize compatibility by integrating with standard stacks like QEMU/KVM, which contrasts with μ-VF's focus on a minimal-overhead, standalone architecture. This highlights a critical gap that μ-VF addresses: providing a holistic virtualization solution for both the software (PS) and reconfigurable hardware (PL, GPIOs) components of an embedded SoC-FPGA.

In contrast, other works propose bespoke operating systems for SoC-FPGAs. FOS [25] is an operating system designed for embedded FPGAs. Similar to μ-VF, it partitions the programmable logic into dynamically reconfigurable zones and employs an on-device hypervisor running on the ARM processor of the SoC. However, the key difference between the two approaches is substantial. In multi-tenant scenarios, FOS follows a task-based model where users submit hardware tasks that the on-board hypervisor schedules and launches on available partial reconfiguration zones. This differs fundamentally from μ-VF's approach: μ-VF assigns complete virtual FPGAs to users, providing persistent slots on both the PS (as containers) and PL (as Lakes), manageable through the abstraction of owning a dedicated virtual FPGA. On the other side, FOS limits tenants to high-level task submission, preventing the low-level hardware/software interactions typical of non-virtualized FPGA development. Without containerized PS execution, tenants cannot deploy isolated software that directly interacts with their PL logic, breaking the tight PS-PL coupling essential for embedded applications. Furthermore, FOS lacks GPIO virtualization, preventing direct access to external peripherals. Despite FOS's focus on modularity, μ-VF demonstrates superior efficiency: on the same ZCU102 platform, μ-VF achieves 20% higher throughput compared to FOS with the same number of tenants. Moreover, while FOS can only allocate 50% of FPGA resources to tenants, μ-VF enables up to 85% resource allocation. Ker-ONE [26] proposes a hypervisor for ARM-FPGA platforms focused on real-time constraints, where the PS is virtualized through traditional virtual machines. However, its usage model remains task-oriented: the system virtualizes accelerators as "virtual devices" shared among VMs, but users cannot load arbitrary hardware designs. Instead, they must select from a pre-defined catalog of accelerators that include built-in preemption support for real-time scheduling. In contrast, μ-VF grants tenants persistent hardware regions (Lakes) where they can deploy any custom logic without restrictions, while also enabling low-latency GPIO access for edge applications.

Finally, it's worth noting higher-level frameworks like ADARE [10], which manage clusters of multiple FPGAs at the edge. ADARE operates at an inter-FPGA orchestration level, making it complementary to μ-VF's intra-FPGA virtualization, which focuses on securely partitioning a single device for multiple tenants.

Table 5. Feature Comparison of Virtualization Frameworks for Embedded SoC-FPGAs

Feature	μ-VF	Ker-ONE [26]	FOS [25]
Primary Goal	Multi-tenancy	Real-time Mgmt.	Modularity
User Abstraction	Complete vFPGA	Task-based	Task-based
PL Management	Arbitrary logic	Pre-defined catalog	Scheduled accel.
PS Management	Container	VM	Host OS user
GPIO Virtualization	Yes (HW)	No	No

6 Conclusions and Future work

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, 20.58% higher than previous approaches.

Future work will focus on: (i) OpenStack [19] 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; and (v) Validation with a broader suite of real-world edge applications to quantify performance impact under specific application-level Service Level Objectives (SLOs).

By demonstrating comprehensive FPGA virtualization within embedded constraints, μ-VF transforms edge FPGAs from single-purpose accelerators into flexible, multi-tenant computing infrastructure. The complete source code for the μ-VF framework is publicly available to facilitate reproducibility and adoption by the research community *https://github.com/vincenzobucaria/u-VF-Enabling-Virtualization-of-Embedded-FPGAs*.

7 Acknowledgements

We used OpenAI's ChatGPT to assist with language fluency and grammar. All technical content and results were developed by the authors.

This work has been supported by the National Science Foundation under grants CCF-2218845,

ECCS-2229472 and ECCS-2329013; by the Air Force Office of Scientific Research under grant FA9550-23-1-0261; by the Office of Naval Research under grant N00014-23-1-2221; and by the Defense Advanced Research Projects Agency under Cooperative Agreement D25AC00374-00.

This work was partially supported by the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU through Project "SERICS – Security and Rights in CyberSpace" under Grant PE00000014.
Approved for public release; distribution is unlimited.

References

[1] Hedi Abdelkrim, Slim Ben Othman, and Slim Ben Saoud. 2017. Reconfigurable SoC FPGA based: Overview and trends. In *2017 International Conference on Advanced Systems and Electric Technologies (IC_ASET)*. IEEE, 378–383.

[2] AMD Xilinx. 2023. DFX Decoupler v1.0 LogiCORE IP Product Guide. https://docs.amd.com/r/en-US/pg375-dfx-decoupler. https://docs.amd.com/r/en-US/pg375-dfx-decoupler PG375.

[3] Christophe Bobda, Joel Mandebi Mbongue, Paul Chow, Mohammad Ewais, Naif Tarafdar, Juan Camilo Vega, Ken Eguro, Dirk Koch, Suranga Handagala, Miriam Leeser, Martin Herbordt, Hafsah Shahzad, Peter Hofste, Burkhard Ringlein, Jakub Szefer, Ahmed Sanaullah, and Russell Tessier. 2022. The Future of FPGA Acceleration in Datacenters and the Cloud. *ACM Trans. Reconfigurable Technol. Syst.* 15, 3, Article 34 (Feb. 2022), 42 pages. doi:10.1145/3506713

[4] Alexander Brant and Guy GF Lemieux. 2012. ZUMA: An open FPGA overlay architecture. In *2012 IEEE 20th international symposium on field-programmable custom computing machines*. IEEE, 93–96.

[5] Ignacio Bravo-Muñoz, Alfredo Gardel-Vicente, and José Luis Lázaro-Galilea. 2020. New Applications and Architectures Based on FPGA/SoC. 1789 pages.

[6] Stuart Byma, J Gregory Steffan, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. 2014. Fpgas in the cloud: Booting virtualized hardware accelerators with openstack. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 109–116.

[7] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. 2022. FPGA HLS today: successes, challenges, and opportunities. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 15, 4 (2022), 1–42.

[8] Salvatore Distefano, Giovanni Merlino, and Antonio Puliafito. 2012. Sensing and Actuation as a Service: A New Development for Clouds. In *2012 IEEE 11th International Symposium on Network Computing and Applications*. 272–275. doi:10.1109/NCA.2012.38

[9] Muhammed Kawser Ahmed, Maximillian Panoff Kealoha, Joel Mandebi Mbongue, Sujan Kumar Saha, Erman Nghonda Tchinda, Peter Esenju Mbua, and Christophe Bobda. 2025. Multi-Tenant Cloud FPGA: A Survey on Security, Trust, and Privacy. *ACM Transactions on Reconfigurable Technology and Systems* 18, 2 (2025), 1–44.

[10] Ian Kersz, Henry Piceni, Michael G Jordan, Jose Rodrigo Azambuja, Fernanda Lima Kastensmidt, and Antonio Carlos S Beck. 2024. ADARE: Adaptive Resource Provisioning in Multi-FPGA Edge Environments. In *2024 37th SBC/SBMicro/IEEE Symposium on Integrated Circuits and Systems Design (SBCCI)*. IEEE, 1–5.

[11] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J Rossbach. 2018. Sharing, protection, and compatibility for reconfigurable fabric with {AmorphOS}. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 107–127.

[12] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. 2020. Do {OS} abstractions make sense on {FPGAs}?. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 991–1010.

[13] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. 2020. A hypervisor for shared-memory FPGA platforms. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 827–844.

[14] Joel Mbongue, Festus Hategekimana, Danielle Tchuinkou Kwadjo, David Andrews, and Christophe Bobda. 2018. FPGAVirt: A novel virtualization framework for FPGAs in the cloud. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 862–865.

[15] Paolo Meloni, Alessandro Capotondi, Gianfranco Deriu, Michele Brian, Francesco Conti, Davide Rossi, Luigi Raffo, and Luca Benini. 2018. NEURAghe: Exploiting CPU-FPGA synergies for efficient and flexible CNN inference acceleration on Zynq SoCs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 11, 3 (2018), 1–24.

[16] Panagiotis Miliadis, Dimitris Theodoropoulos, Dionisios Pnevmatikatos, and Nectarios Koziris. 2024. Architectural support for sharing, isolating and virtualizing FPGA resources. *ACM Transactions on Architecture and Code Optimization* 21, 2 (2024), 1–26.

[17] Carlos Moratelli, Sergio Johann, Marcelo Neves, and Fabiano Hessel. 2016. Embedded virtualization for the design of secure IoT applications. In *Proceedings of the 27th International Symposium on Rapid System Prototyping: Shortening the*

Path from Specification to Prototype. 2–6.

[18] Anna Panagopoulou, Michele Paolino, and Daniel Raho. 2023. Virtio-FPGA: a virtualization solution for SoC-attached FPGAs. In *2023 IEEE International Conference on Electrical Systems for Aircraft, Railway, Ship Propulsion and Road Vehicles & International Transportation Electrification Conference (ESARS-ITEC)*. IEEE, 1–6.

[19] Tiago Rosado and Jorge Bernardino. 2014. An overview of openstack architecture. In *Proceedings of the 18th international database engineering & applications symposium*. 366–367.

[20] Nicolas Serrano, Gorka Gallardo, and Josune Hernantes. 2015. Infrastructure as a service and cloud technologies. *IEEE software* 32, 2 (2015), 30–36.

[21] Douglas J Smith. 1998. *HDL Chip Design: A practical guide for designing, synthesizing and simulating ASICs and FPGAs using VHDL or Verilog*. Doone publications.

[22] Mattia Tibaldi and Christian Pilato. 2023. A survey of FPGA optimization methods for data center energy efficiency. *IEEE Transactions on Sustainable Computing* 8, 3 (2023), 343–362.

[23] Ramao Tiago Tiburski, Carlos Roberto Moratelli, Sergio F Johann, Marcelo Veiga Neves, Everton de Matos, Leonardo Albernaz Amaral, and Fabiano Hessel. 2019. Lightweight security architecture based on embedded virtualization and trust mechanisms for IoT edge devices. *IEEE Communications Magazine* 57, 2 (2019), 67–73.

[24] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. 2018. A survey on FPGA virtualization. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 131–1317.

[25] Anuj Vaishnav, Khoa Dang Pham, Joseph Powell, and Dirk Koch. 2020. FOS: A modular FPGA operating system for dynamic workloads. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 13, 4 (2020), 1–28.

[26] Tian Xia, Ye Tian, Jean-Christophe Prévotet, and Fabienne Nouvel. 2019. Ker-ONE: A new hypervisor managing FPGA reconfigurable accelerators. *Journal of Systems Architecture* 98 (2019), 453–467.

[27] Sadegh Yazdanshenas and Vaughn Betz. 2017. Quantifying and mitigating the costs of FPGA virtualization. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–7.

[28] Yue Zha and Jing Li. 2020. Virtualizing FPGAs in the cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 845–858.

[29] Yue Zha and Jing Li. 2021. Hetero-ViTAL: A virtualization stack for heterogeneous FPGA clusters. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 470–483.

[30] Jiansong Zhang, Yongqiang Xiong, Ningyi Xu, Ran Shu, Bojie Li, Peng Cheng, Guo Chen, and Thomas Moscibroda. 2017. The Feniks FPGA operating system for cloud computing. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*. 1–7.

Received July 2025; revised September 2025; accepted October 2025