SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Frontend Development of Distributed FPGA Management in Serverless Computing

Zirong Cai

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Frontend Development of Distributed FPGA Management in Serverless Computing

Frontend-Entwicklung von verteiltem FPGA-Management im Serverless Computing

Author:	Zirong Cai
Supervisor:	Bhatotia Pramod; Prof. Dr.-Ing.
Advisor:	Koshiba Atsushi; Dr. Ph.D., Chen Jiyang; M.Sc.
Submission Date:	15.04.2023

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.04.2023 Zirong Cai

Acknowledgments

Abstract

In recent years, Field Programmable Gate Arrays (FPGAs) have gained prominence as a potential solution for enhancing the performance of cloud-based workloads. Nonetheless, their integration into cloud environments is impeded by inherent hardware limitations, including the absence of resource management and isolation support. Consequently, there is no existing commercial multi-tenant solution for distributed FPGA management, which leads cloud providers to rely on per-VM static binding of FPGA resources through PCI-passthrough mechanisms. Moreover, FPGA workloads display erratic behavior and extended periods of underutilization, which can be alleviated by enabling the sharing of FPGAs among multiple tenants.

In this thesis, we propose a novel framework for distributed FPGA sharing designed to accelerate serverless unikernel applications within cloud environments. This framework incorporates OpenFaaS, Kubernetes, vAccel Urunc, and Funky Monitor into a cohesive system. Our evaluation reveals that the performance of the newly developed framework is comparable to that of the original Funky Monitor deployed application, showcasing the potential of this framework to address FPGA-related challenges in cloud environments.

Contents

1 Introduction

In the past decade, cloud computing has become the preeminent technology for developing, deploying, and maintaining complex infrastructures and services at scale. This technology facilitates on-demand resource consumption in a multi-tenant environment, necessitating the adoption of a cloud-native approach for dynamic performance scaling. To address the performance demands of compute-intensive workloads that current CPUs cannot meet, heterogeneous computing has gained prominence as a means to uphold service level agreements (SLAs) in the cloud. Consequently, cloud vendors offer Field Programmable Gate Arrays (FPGA)-based compute instances (e.g., AWS F1 instances) alongside GPU-enabled instances.

FPGAs have emerged as a transformative technology, empowering users to create custom hardware specifically tailored to distinct computational tasks. This highly adaptable innovation has revolutionized the performance of various cloud workloads, including machine learning [47, 48, 55], databases [12, 26, 39], network stacks [56], storage stacks [35], and graph processing [5, 17, 42]. Compared to traditional host-centric computing, FPGAs have demonstrated significant performance improvements [13, 45].

However, to deploy FPGAs for cloud applications, it is necessary to design hardware accelerators that not only meet latency requirements but also improve throughput[1]. Minimizing latency is crucial, as requests from external networks arrive at unpredictable rates and typically cannot be batched. Furthermore, request unpredictability may result in FPGA underutilization, making the reservation of one FPGA for each service requiring it an inefficient use of resources. From a cloud provider's perspective, sharing the FPGA among multiple tenants could serve as a potential solution for enhancing resource utilization.

Currently, cloud providers utilize the PCI-passthrough mechanism to support FPGAs [49], which is unsuitable for multi-tenant environments as it dedicates an FPGA device to a single VM instance. This approach results in underutilization of FPGA resources and impedes resource consolidation [41, 46]. The primary challenge in providing multi-tenant support for distributed FPGAs stems from the lack of requisite hardware support for resource isolation and management, such as memory management units (MMUs) or context switching [32, 44, 46]. Recent studies have addressed these limitations by introducing hardware support called Shell—a static set of I/O and control logic configured within a portion of the FPGA [27, 32]—which enables tempo-

ral/spatial sharing of reconfigurable slots, I/O multiplexing, and memory protection. However, the Shell-based approach is only suitable for multiplexing an FPGA board on a single node and does not support multiple nodes in distributed cloud environments without system software support [54].

To manage distributed architectures where FPGA boards are distributed among multiple nodes and different Shells are integrated into FPGAs, Funky, a unikernel-based FPGA management framework is proposed. Designed to be Shell-agnostic, Funky can be built upon any Shells developed by either FPGA vendors [21, 52] or academic projects [27, 32]. The Funky framework operates across distributed nodes and manages available reconfigurable FPGA slots while collaborating with the underlying Shells. Funky's unikernel architecture separates FPGA applications from OSes/VM instances, encapsulating each application within a unikernel and executing them directly on the hypervisor. This architecture allows applications to be deployed or migrated on any distributed node with minimal performance penalties, efficiently utilizing available FPGA resources.

With the help of Funky, the serverless computing paradigm can be harnessed to enable fine-grained FPGA sharing among tenants, thereby improving resource utilization. Serverless computing [7] is an architectural pattern for cloud applications that delegates server management to the cloud provider. Users deploy each application functionality as a function, which is then scheduled, executed, scaled, and billed according to the precise requirements of the moment. This approach allows cloud-native applications to leverage FPGAs for accelerating compute-intensive workloads such as neural network inference, web searches, and image processing seamlessly.

Although the Funky framework has been developed to facilitate the deployment or migration of unikernel applications, building serverless functions requires further advancements in this domain. This thesis aims to address the limitations of the current state-of-the-art by proposing a comprehensive solution that eases the build and deploy process of unikernels, allowing developers to utilize Funky at a higher level. The primary goal is to enable a more efficient and streamlined approach to the development and deployment of serverless functions that harness the power of FPGA technology.

A key component of the proposed solution is Urunc, an extension of the Kata Container[1] designed for deploying unikernel applications. Urunc is fully Container Runtime Interface (CRI) compatible [33] and can be integrated into the Kubernetes ecosystem [10]. By combining the power of Urunc, the OpenFaaS framework [20], and Kubernetes, this thesis presents a serverless framework that allows developers to write functions that utilize FPGA boards as accelerators. The resulting system offers significant benefits in terms of increased efficiency, reduced costs, decreased energy consumption, and more.

The organization of this thesis is structured as follows: Section 2 furnishes a compre-

hensive background on the essential concepts underpinning Field-Programmable Gate Arrays (FPGAs), unikernels, the Funky framework, and the urunc extension, laying the foundation for understanding the context and challenges addressed throughout the thesis. Section 3 delves into the details of the Funky framework and its seamless integration with Kubernetes and OpenFaaS. Section 4 introduces the design goal and the proposed solution, encompassing the urunc extension and the implementation of a watchdog mechanism for serverless functions, presenting the novel aspects and design choices of our approach. Section 5 outlines the implementation process, describing how urunc is ported into the Funky environment and how we implemented our watchdog function. Section 6 showcases the experimental results and performance evaluation, highlighting the efficacy and advantages of the proposed solution in the context of FPGA-accelerated serverless computing. Section 7 discusses related work in the field, drawing connections and differentiating our thesis from existing research. Section 8 provides a conclusion, summarizing the key contributions and findings of the thesis, while Section 9 offers potential directions for future research, pointing out areas where further exploration and development could be beneficial.

2 Background

2.1 FPGA Architecture and Programming Model

Our research focuses on the utilization of PCIe-connected FPGAs, a prevalent deployment model in cloud computing environments [2, 15]. The architecture of these FPGAs comprises a dynamic region and a static region, commonly referred to as the Shell [22, 53]. The dynamic region is characterized by its reconfigurability, allowing users to construct their hardware logic during runtime. The static region, the Shell, encompasses the necessary glue logic that facilitates the connection of accelerators in the dynamic region to the external world. This glue logic encompasses interfaces with the host system, interconnections with onboard devices such as DDR and network ports, as well as Direct Memory Access (DMA) modules. Additionally, the Shell is responsible for providing an interface for the reconfiguration of the dynamic region.

With regards to the programming model, it is split between the host system (CPU) and the device (FPGA). The host programming model enables task offloading onto the FPGA, while the device programming model represents the actual kernel code. Our research targets OpenCL as the host programming model, which is an open standard for parallel programming of heterogeneous devices [36]. OpenCL offers a platform-agnostic programming model, and is widely adopted by FPGA vendors such as Xilinx and Intel FPGAs.[53, 23]. In the OpenCL framework, heterogeneous cores, including CPUs, GPUs, and FPGAs, are abstracted as devices. OpenCL code consists of two components: the OpenCL host code that runs on the CPU (host), and the kernel code that runs on the device (kernel). The host code is responsible for initializing buffers in the device memory and submitting commands for data transfers and kernel executions between the host and the device. This is achieved through the OpenCL host APIs, which allow the host code to launch kernels on the target device and manage device-side memory.

As for the device programming model (FPGA), our work operates as an abstraction layer for the host code, and therefore imposes no limitations on the programming of the kernel code. All platform-dependent programming languages are supported, including Hardware Description Languages (HDL) [8, 19], High-Level Synthesis (HLS) [6, 24, 30, 36, 3], and Domain-Specific Languages (DSLs) [7, 14, 31, 38].

2.2 Unikernel

Unikernel has emerged as an innovative approach to lightweight virtualization, offering numerous advantages such as enhanced security, performance, and resource efficiency. Unikernels are specialized, single-address space images that incorporate an application, its configuration, and application-specific operating system (OS) components. Utilizing library OSes, unikernels facilitate direct linkage to a single application, creating a lightweight virtualization environment. The primary goals of unikernel-based virtualization are to improve security, performance, and resource efficiency [9, 29, 34, 37].

Hardware accelerators, such as GPUs and FPGAs, have become increasingly popular in cloud environments. However, unikernels have not yet offered support for these accelerators [16]. Contemporary unikernels provide limited hardware device support, generally confined to block devices [9, 34] and network devices [37], due to their early-stage development and restricted functionality. To date, no unikernels have been found to support hardware accelerators. Adapting existing FPGA applications or frameworks to unikernels is not an easy task.

Our design is based on IncludeOS [9], a prominent unikernel designed to provide a lightweight, specialized virtualization solution that caters to the demands of modern cloud computing environments. Developed using C++, IncludeOS is particularly suited to bridge the gap between clean-state and POSIX-like unikernel families, offering a versatile and efficient platform for application-specific virtualization. As a unikernel, IncludeOS incorporates only the necessary OS components required for a particular application, leading to a lean and secure runtime environment.

One of the key features of IncludeOS is its support for C/C++ applications through a custom GCC toolchain. This compatibility enables developers to leverage existing programming expertise and take advantage of the performance benefits offered by C/C++ when building unikernel applications. Furthermore, IncludeOS is designed to be platform-agnostic, supporting a wide range of hypervisors, such as KVM, Virtualbox, and VMware, on x86 architectures. This flexibility allows developers to deploy IncludeOS-based unikernels across various cloud platforms and environments, further promoting the portability and scalability of unikernel applications.

In the case of KVM, IncludeOS not only supports QEMU/KVM but also Solo5 [51], a lightweight, sandboxed execution environment that complements IncludeOS in providing an efficient and secure virtualization solution. Solo5 serves as a thin hypervisor layer, introducing its own application binary interface (ABI) that facilitates communication between the unikernel and the underlying hardware. It implements a PIO/MMIO-like interface for asynchronous I/O, utilizing a poll hypercall to handle I/O operations efficiently. This approach ensures minimal overhead and reduced

latency when executing I/O-bound applications.

The combination of IncludeOS and Solo5 offers several notable benefits for unikernel-based virtualization. By leveraging the strengths of both technologies, the resulting virtualization solution exhibits improved boot times, reduced memory footprint, enhanced security, and streamlined deployment processes. As a result, the IncludeOS and Solo5 tandem represents a powerful and versatile platform for building and deploying unikernel applications, especially in resource-constrained and security-sensitive cloud environments [51].

2.3 Funky Unikernel

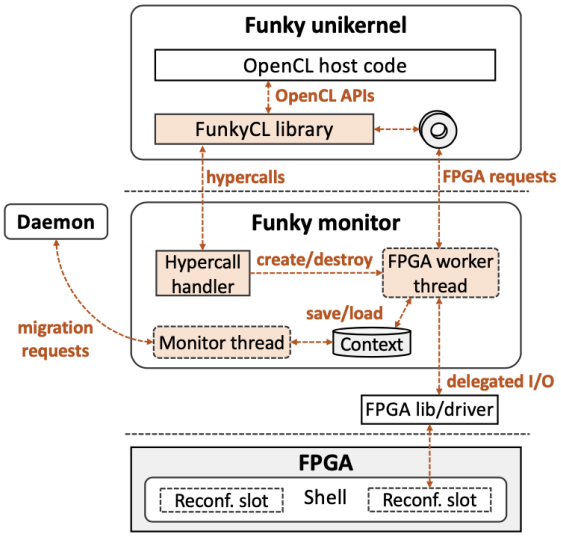

Figure 2.1: Funky's unikernel architecture

Funky is a unikernel-based architecture for distributed FPGA management targeting cloud environments. It's platform-agnostic, supports lightweight unikernel-based FPGA management and distributed resource management.

Funky provide a customized UKVM monitor, called Funky monitor to allow unikernels to access the FPGA devices using a set of hypercalls. Funky monitor is a user process serving as a thin hypervisor layer for unikernels. The Funky monitor is responsible for reserving an isolated context and launching the corresponding unikernel. It has two core functionalities: hypercalls and FPGA worker thread. The hypercalls allows the Funky unikernel to get an available FPGA slot and reconfigure the obtained

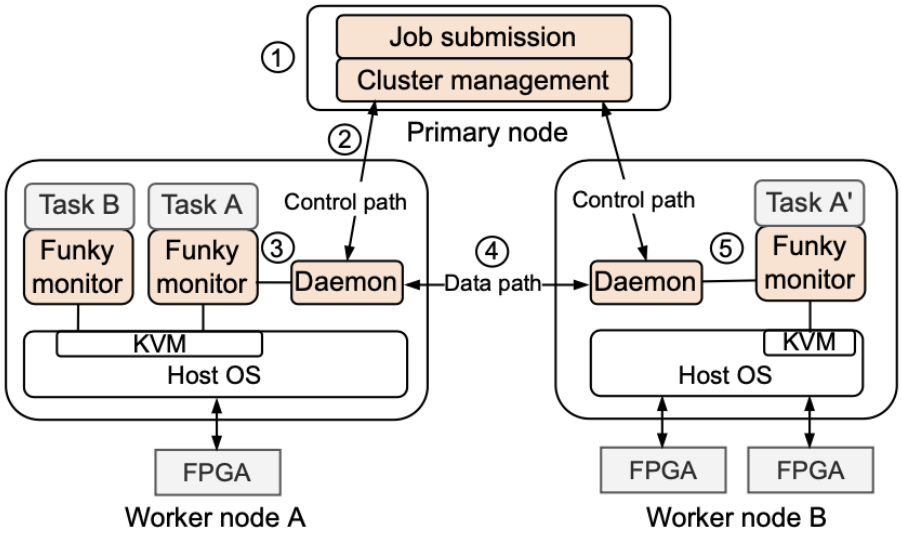

Figure 2.2: Funky's distributed framework

slot by transferring a bitstream to the Funky monitor. Meanwhile, a FPGA worker thread will be created to handle the asynchronous FPGA-related requests from the Funky unikernel.

To achieve application portability and programmability, Funky provides FunkyCL, which is an OpenCL-compatible library. FunkyCL is fully compatible with the native OpenCL specification standardized by Khronos group[28], meaning developers does not need to make any changes to their OpenCL host code, except for changes due to vendor-specific OpenCL extensions or functional limitations of unikernels.

Funky also supports flexible FPGA task management. Funky unikernels using FPGAs can be paused or migrated across distributed nodes. In the first situation, the FPGA state is reserved locally so that the suspended unikernel can immediately resume its execution on a local node. These works are done by a new thread spawned by the Funky monitor. It can receive and handle commands from the scheduler through inter-process communication. On the other hand, upon receiving a migration request, the monitor thread captures the vCPU context and saves the guest's dirty pages, vCPU state, and FPGA state into a single file. The file is then transferred to the target node over the network, where it is restored and resumed to continue the execution.

In conclusion, funky-unikernel is a unique framework that enables the lightweight management of FPGA resources with the help of unikernels. By leveraging the advantages of unikernels, such as small size and quick boot time, funky can efficiently manage

FPGA resources without the overheads associated with traditional virtualization. Additionally, funky's hypervisor-based migration mechanism and checkpointing features further enhance the platform's flexibility and reliability. Overall, funky-unikernel has the potential to greatly simplify the management of FPGA resources, making it an exciting development for FPGA-based applications.

2.4 Kata Container

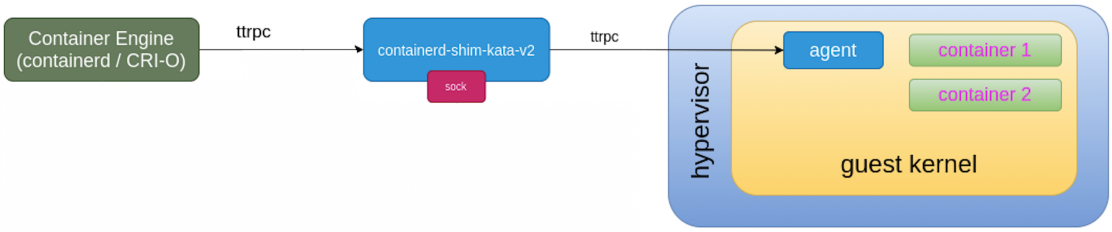

Figure 2.3: Kata Container 2.x Architecture

Kata Containers is an open-source project that provides a secure and lightweight container runtime environment. It is designed to combine the speed and agility of containers with the security of virtual machines (VMs) through hardware virtualization. Kata Containers creates a lightweight VM for each container, with its own kernel, that runs directly on top of the host's hardware.

The idea behind Kata Containers is to bring the benefits of both containers and VMs together. Containers are lightweight and agile, providing fast startup times and low resource utilization. However, containers share the same kernel with the host, which means that if a container is compromised, it can potentially affect the host and other containers running on the same host. On the other hand, VMs provide better isolation and security, but they are heavyweight and have a slower startup time.

Kata Containers addresses these issues by running each container in its own lightweight VM, providing the same level of isolation and security as a VM, but with the same fast startup time and low resource utilization as a container. This approach allows containers to be used in environments where security is a top priority, such as multi-tenant environments, without sacrificing performance.

One of the main differences between Kata Containers and other container runtimes, such as Docker or rkt, is that it uses hardware virtualization to create the VMs, rather than relying on container isolation features provided by the Linux kernel. This means that Kata Containers can run on any hardware platform that supports hardware virtualization, without relying on specific kernel features.

Another advantage of Kata Containers is that it supports a wide range of container image formats, including Docker images, Open Container Initiative (OCI) images, and Kubernetes container images. This means that existing container workloads can be easily migrated to Kata Containers without any changes to the application code or the container image.

However, there are also some disadvantages to using Kata Containers. One is that it requires hardware virtualization support, which may not be available on all hardware platforms. Another is that creating a VM for each container can result in a higher memory overhead compared to traditional container runtimes.

In summary, Kata Containers provides a secure and lightweight container runtime environment that combines the benefits of both containers and VMs. By running each container in its own lightweight VM, Kata Containers provides the same level of isolation and security as a VM, but with the same fast startup time and low resource utilization as a container. While it has some disadvantages, such as the requirement for hardware virtualization support, the benefits of using Kata Containers make it a compelling option for organizations looking to run container workloads in a secure and efficient manner.

2.5 vAccel Urunc

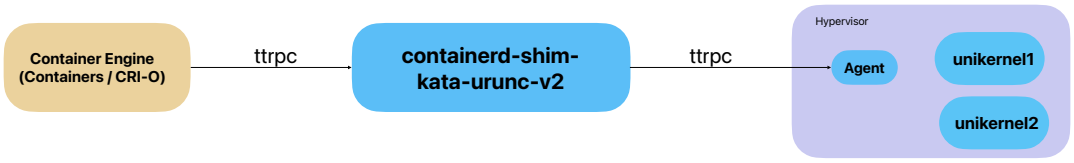

Figure 2.4: urunc Architecture

Urunc is an extension of Kata Containers that provides the capability to deploy unikernels instead of traditional containers. Unlike Kata container who creates a lightweight VM for each container for better isolation and security, unikernels alone offer a more lightweight and secure approach to application deployment by packaging the application and the operating system into a single executable image.

Urunc achieves this by packing the unikernel binary into a Docker image, which allows for easy distribution and deployment using the existing Docker ecosystem. Upon deployment, Urunc unpacks the image and extracts the unikernel binary, which is then executed using the Solo5-hvt hypervisor.

Urunc offers several benefits over traditional container deployment. The most outstanding one is that, unikernels are much more lightweight than traditional containers, as they only contain the application and operating system components that are required for the application to run. This results in a smaller image size and lower resource utilization, which in turn leads to better performance and scalability.

Overall, Urunc provides a powerful and flexible platform for deploying unikernels using the existing Docker ecosystem. By leveraging the Solo5-hvt hypervisor, Urunc is able to provide a lightweight, secure, and efficient platform for running unikernels.

2.6 Summary

The implementation of a serverless framework requires a mechanism to share FPGA resources among multiple users, and the innovative use of the Funky unikernel offers a promising solution. With the ability to enable the sharing of a FPGA board, Funky unikernel serves as the backend of the serverless framework. This novel approach has the potential to transform the field of serverless computing by offering a more cost-effective approach to deploying and managing FPGA resources. Additionally, the lightweight and secure nature of unikernel further enhances its potential to revolutionize the field of serverless computing.

Moreover, Urunc, an extension of the Kata Containers project, bridges the gap between container and unikernel, and allows for the deployment of unikernel in a container-like manner. This eliminates the need for users to learn new tools and workflows, as the deployment process is similar to that of containerized applications. By packaging the unikernel binary into a Docker image and deploying it using Solo5-hvt as the hypervisor, Urunc leverages the infrastructure provided by Kata Containers, making unikernel deployment accessible to a wider audience.

In conclusion, the combination of Funky unikernel and Urunc provides a comprehensive solution for FPGA resource sharing and deployment. The containerization-based unikernel deployment offers a simpler and more accessible workflow, and has the potential to greatly reduce the cost and complexity of FPGA-based serverless computing. This innovative solution has the potential to transform the field of serverless computing and accelerate the development of FPGA-based applications.

3 Overview

3.1 System Goals

The goal of this project is to design a novel runtime framework that enables serverless functions to seamlessly use Field-Programmable Gate Arrays (FPGAs) without requiring users to have specialized knowledge of FPGA hardware or programming. The framework should also support existing serverless frameworks, such as OpenFaaS and Kubernetes, to ensure compatibility and ease of integration with existing systems.

To enable efficient event-driven execution, the framework should accommodate an event-driven execution model. Furthermore, the framework should present serverless APIs that abstract the intricacies of FPGA programming, thus rendering the framework more user-friendly and accessible to developers without FPGA proficiency.

Resource efficiency holds paramount importance for the framework, and to accomplish this, time- and space-sharing techniques should be implemented on the FPGA. This guarantees effective FPGA utilization, avoiding wastage, and enabling more efficient resource allocation.

Furthermore, the framework should have low latency, which is imperative for real-time applications. To achieve fast FPGA reconfiguration and minimize latency, the framework should exploit advanced FPGA features, such as partial reconfiguration or schedule similar logics on the same FPGA.

Overall, the proposed runtime framework aims to address the challenges associated with FPGA-based serverless computing by providing a user-friendly and efficient solution that seamlessly integrates with existing serverless frameworks while achieving low latency and high resource efficiency.

Currently, our system features only the Funky Monitor as its backend, facilitating the deployment of unikernel applications that utilize FPGA boards. The integration of a runtime application (e.g., runc in the container realm) is vital for simplifying the development and deployment of unikernel applications within the Funky Monitor. Following this step, the incorporation of a platform such as Kubernetes or an alternative orchestrator becomes imperative for the monitoring and scheduling of all unikernel applications, ultimately improving the system's overall efficiency. Notably, the final aspect, which involves deploying OpenFaaS to Kubernetes, is already in place. This constitutes the primary emphasis of our thesis.

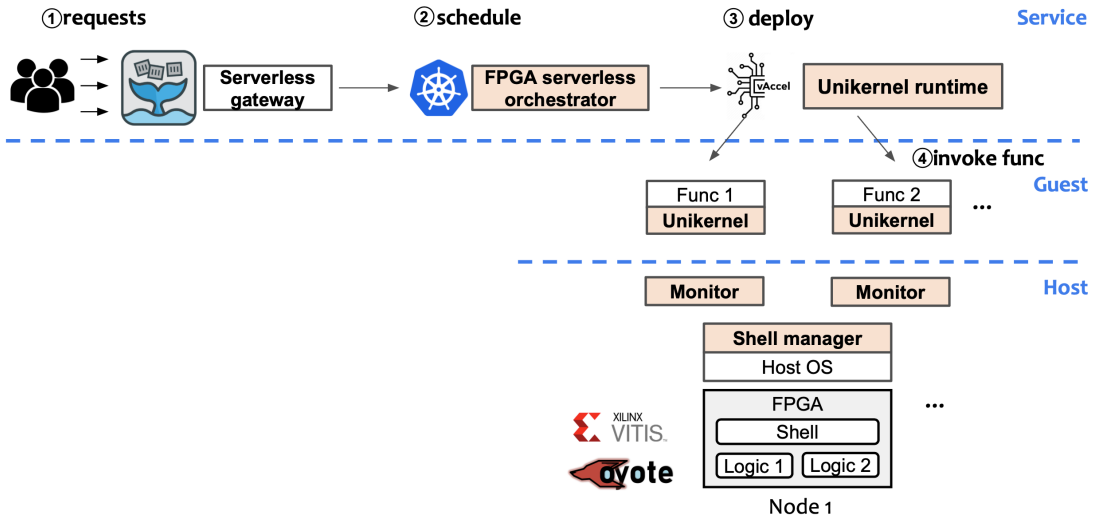

Figure 3.1: System Design

3.2 System Overview

The proposed system architecture in this thesis employs Kubernetes as the orchestrator for managing nodes and deploys OpenFaaS within the Kubernetes cluster to enable serverless function capabilities. Furthermore, the architecture utilizes Urunc as the unikernel runtime for building and deploying unikernel applications.

In the first layer, developers utilize OpenFaas to deploy unikernel applications, while users send requests to the OpenFaas Gateway to trigger the unikernel function, thereby initiating the application execution process.

The second layer is responsible for efficient resource allocation and load balancing. Kubernetes schedules the unikernel application to an appropriate node based on the availability of resources, ensuring optimal performance.

The third layer involves deploying the unikernel application using the ukvm hypervisor through the Vaccel Urunc framework. Additionally, the application is integrated with a Watchdog, which listens to user requests and initiates the unikernel function only when a request is received. This event-driven execution model minimizes resource wastage and ensures efficient execution.

After the unikernel application is deployed and initialized, it runs inside the node and awaits user requests for execution. Overall, this system architecture provides a seamless and efficient solution for deploying and executing unikernel applications using OpenFaas and Kubernetes, while ensuring resource efficiency and event-driven execution.

3.3 System Workflow

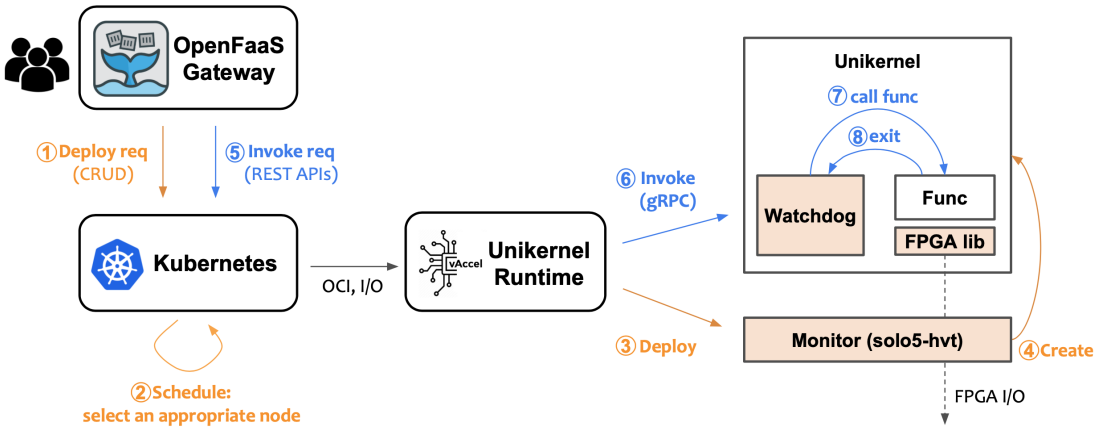

Figure 3.2: System Workflow

The system workflow encompasses several stages for managing user requests and executing unikernel applications, employing OpenFaaS and Kubernetes.

Initially, the user submits an HTTP request to the OpenFaaS Gateway. In the subsequent stage, the OpenFaaS Gateway relays the request to Kubernetes, which determines the application's location. Kubernetes then forwards the request to the watchdog operating within the unikernel application to initiate the function.

In the third stage, the function is executed, and the Funky Monitor reconfigures the FPGA board utilizing the user-specified bitstream. If no slots are available on the FPGA board, the application may either wait or migrate to another node with unoccupied FPGA slots.

The final stage involves generating and transmitting the response. Upon the FPGA's execution, the Funky Monitor conveys the result from the FPGA board back to the host function, which subsequently forwards it to the user via HTTP.

3.4 Simplified User Interface

After the successful implementation of the framework, users are now able to develop functions that harness FPGA boards as accelerators. Nonetheless, the entry barrier for deploying an FPGA kernel and configuring the FPGA board continues to be considerably steep. Consequently, this thesis endeavors to abstract intricate aspects of FPGA boards, such as reconfiguring the board and data transfer processes, while

offering a streamlined user interface that facilitates more straightforward and efficient utilization of FPGA boards.

4 Design

4.1 kata container

Kata Containers is a lightweight, open-source container runtime that leverages hardware virtualization technologies to provide a unique solution for running containers with the benefits of both virtual machines (VMs) and containers. By running each container in its own lightweight VM, Kata Containers provides greater security and workload isolation than traditional containers while still maintaining the speed and efficiency of containerization.

The framework of Kata Containers 2.x is comprised of two main components: the Agent and the Runtime. The Agent is a long running process that runs inside the VM, it acts as the supervisor for managing the containers and the workload running within those containers. When a user requests the creation of a container the agent creates a container environment in the container specific directory and then spawns the workload inside the container environment. The Kata Containers runtime (the containerd-shim-kata-v2 binary) is a shimv2 compatible runtime.The runtime is responsible for starting the hypervisor and it's VM, and communicating with the agent using a ttRPC based protocol over a VSOCK socket that provides a communications link between the VM and the host.This protocol allows the runtime to send container management commands to the agent.

Overall, the Kata Containers system is designed to provide a secure and isolated runtime environment for containers. Its architecture allows it to provide hardware-level isolation and security, while also leveraging the existing Docker ecosystem for container image management. The system's compatibility with existing container orchestration systems also makes it easy to integrate into existing container-based infrastructures.

4.2 vAccel urunc

Urunc is a specialized implementation of the Kata container technology that has been developed to enable the deployment of unikernel applications. The system provides a runtime environment that is capable of deploying unikernels directly on a supported hypervisor, such as solo5-hvt. This novel approach offers several advantages, such as

enhanced performance and a reduction in resource utilization, which make Urunc an optimal solution for deploying unikernel applications.

The main distinguishing factor between Urunc and Kata container is found in the runtime components, specifically the containers-shim-kata-v2. When tasked with launching a unikernel application, Urunc's runtime component deploys the unikernel directly onto the hypervisor, bypassing the need to run the application within a virtual machine for security purposes. This approach is notably faster, lighter, and more resource-efficient. The inherent security of unikernels obviates the need for virtual machines, further bolstering the advantages of Urunc's approach.

Furthermore, Urunc also provides users with an image builder tool, which allows the packaging of the unikernel binary into a container image. Subsequently, the binary can be extracted from the image and executed by mapping the rootfs/disk of the unikernel instance to the container image rootfs with the assistance of devmapper snapshotter. This attribute simplifies the deployment of unikernel applications by enabling users to use the same deployment methods as those employed for deploying container applications, without requiring any additional comprehension of unikernels.

4.3 Watchdog for funky unikernel

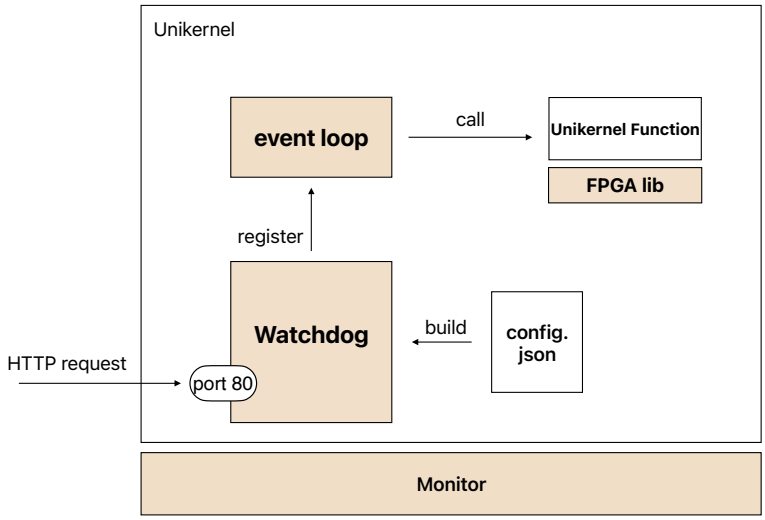

Figure 4.1: Watchdog Design

A watchdog is a crucial component responsible for the starting and monitoring

of functions in a system. It acts as an "init process" and is equipped with an embedded HTTP server, which enables it to support concurrent requests, timeouts, and health checks. As an interface between the outside world and the function, the watchdog is responsible for starting a new process for each request and using STDIO for communication.

For every function to work seamlessly with the watchdog, it needs to embed the watchdog binary and use it as its ENTRYPOINT or CMD. This makes the watchdog the init process for the container, which allows it to pass in the HTTP request via stdin and read the HTTP response via stdout. Therefore, the process itself does not need to have any knowledge about the web or HTTP, as the watchdog handles all the necessary communication between the process and the outside world.

To achieving event-driven execution of a unikernel function, the main part is to integrate the unikernel application into the watchdog process during the build process to ensure its availability and functionality upon deployment. The watchdog process is designed to utilize the net library of includeOS to listen to a port for incoming requests from clients. Once a valid request is received, the watchdog process will call the appropriate unikernel function to configure the FPGA board, transfer data to the board, and execute the unikernel application. Upon retrieving the result, the watchdog process will send it back to the client.

As of current research, a very rough version of the watchdog has been implemented. This version of the watchdog waits in a while loop for client's request, when a request comes in, it directly calls the user-defined unikernel function. In subsequent iterations, the watchdog will be refined to include additional functionality, such as supporting for concurrent requests and health checks, to ensure that event-driven execution is reliable and efficient.

4.4 FPGA Programming Interface

In the traditional approach, developers seeking to utilize FPGA for accelerating their functions must first consider writing the kernel bitstream, locate the FPGA device, reconfigure it, transfer data to the device, execute the kernel function, and retrieve the result. This complex process can be overwhelming for most cloud users, who desire a simpler method to expedite their functions while minimizing costs.

To address this need, we have designed a programming interface that allows developers to leverage FPGA boards by merely providing specific information in JSON format. This includes the bitstream name or URL for downloading a bitstream file, input data, and the expected output data. Our system maintains a database of common bitstreams that users can directly access by specifying the bitstream name in the JSON

file. Alternatively, users can employ a customized bitstream by uploading the file to the internet and providing the URL in the JSON file. This streamlined approach significantly simplifies the process of utilizing FPGA boards for function acceleration, catering to the preferences of cloud users.

```
{
  "fpga_configuration": {
    "bitstream_name": "example_bitstream",
    "bitstream_url": "https://example.com/path/to/bitstream/file.bit"
  },
  "input_data": {
    "data_1": 42,
    "data_2": 78,
    "data_3": 21
  },
  "expected_output_data": {
    "output_1": 105,
    "output_2": 37
  }
}
```

Listing 4.1: bitstream_config.json file

5 Implementation

5.1 Porting Urunc to the Funky Environment

The goal of this section is to detail the steps taken to adapt the Urunc tool, originally designed for deploying unikernels, to support unikernels utilizing FPGA boards in the Funky environment. The following subsections outline the necessary changes made to Urunc to enable compatibility with the specific requirements of the Funky environment.

5.1.1 Making Xilinx Available for Unikernel Applications

Initially, Urunc was only designed to deploy unikernels without FPGA board support. To accommodate unikernels that utilize an FPGA board, we first needed to make Xilinx available for the unikernel applications. Since Urunc deploys the unikernel using the ukvm on the host, we achieved this by adding the environment setting of Xilinx into the command execution environment. This change allows unikernel applications to access Xilinx resources, effectively extending Urunc's capabilities to support unikernels with FPGA boards.

5.1.2 Automating Bitstream File Specification

To execute a Funky unikernel application, the bitstream file is required. While the bitstream file is attached via diskbuilder in the CMakefile, its name must still be specified during execution. We modified Urunc to automate this process by implementing a mechanism for detecting the bitstream name.

The modified Urunc requires users to create a file named *name_of_bitstream.xclbin* inside the data directory while building the Docker image. Urunc will then navigate to the directory, locate the bitstream file, and append its name to the execution command. This approach ensures that the Funky environment is aware of the correct bitstream name during the unikernel application execution.

5.1.3 Addressing Filesystem Incompatibilities

The original implementation of Urunc supported the ext2fs filesystem exclusively, whereas IncludeOS utilized ext4. This discrepancy led to issues with mounted di-

rectories not being unmounted successfully and, consequently, being unable to be deleted. To resolve this problem, we analyzed the error messages and adapted the directory paths according to the updated filesystem requirements. By addressing these incompatibilities, we ensured a seamless integration between Urunc and the Funky environment, allowing for efficient deployment of unikernel applications with FPGA board support.

5.2 Watchdog Implementation

The implementation of the watchdog subsystem aims to facilitate the integration of the funky application to the serverless world. This section provides an overview of the main components involved in the watchdog's operation and addresses the challenges encountered during its implementation.

5.2.1 Input Argument of Watchdog

The watchdog receives the bitstream name from Urunc, which is necessary for the execution of the funky application. This information is crucial for the proper functioning of the subsystem, as it allows the watchdog to locate and execute the application in response to incoming user requests.

5.2.2 TCP Server Configuration and Requests Handling

To create a network stack, the watchdog leverages the Super_stack module. It sets up a TCP server on port 80 and adds a TCP connection handler to manage incoming client requests.

The TCP connection handler is responsible for processing incoming client requests. The system only accepts GET requests; if a GET request is received, the handler invokes the funky function implemented within the watchdog using the same *argc* and *argv* as the watchdog itself. The result is then sent back to the client with the close TCP field set. In cases where a different request type is received, the handler responds with a 404 error.

5.2.3 Maintaining the Handler with IncludeOS

The handler's maintenance is handled by IncludeOS, which ensures that the watchdog function runs periodically without reaching an endpoint. However, a challenge arises during this process: the *argc* and *argv* of the watchdog are lost. To overcome this issue,

a temporary variable is defined to store these two variables, and this variable is passed to the handler's lambda function for use during execution.

5.2.4 Network Configuration

The network configuration of the virtual machine can reside in a JSON file, named config.json, placed in the same folder of the unikernel application. It should look like as following depending on your need:

```
{
  "net" : [
    {
      "iface": 0,
      "config": "dhcp-with-fallback",
      "address": "10.0.0.42",
      "netmask": "255.255.255.0",
      "gateway": "10.0.0.1"
    }
  ]
}
```

Listing 5.1: config.json file

The parameter "iface" refers to the identification number of the interface, while "config" denotes the type of configuration being applied. Specifically, in this context, the configuration type is set to "dhcp-with-fallback", which entails the attempt to acquire an IP address via Dynamic Host Configuration Protocol (DHCP). If this attempt fails, the specified IP address is used as a fallback. Other configuration options that can be used include "dhcp" and "static". The "address" parameter specifies the IP address to be used in the event that the DHCP protocol fails to acquire one. Furthermore, "netmask" indicates the netmask to be utilized in the network, and "gateway" denotes the default gateway IP address to be used for network communication.

5.2.5 Funky Function Adaptation for Testing

For testing purposes, the funky function's name is manually changed from "main" to another name, and the new function is called within the watchdog function. In future implementations, this process should be automated by Urunc, allowing users to execute their applications without the need for manual intervention in changing function names. This enhancement will streamline the process and improve the overall user experience.

6 Evaluation

The objectives of this experimental study aim to determine if our framework generates a tolerable overhead in comparison to native execution, which is directly facilitated by Funky.

6.1 General Information

In this section, I will display the environment of the performance test. The test program is running on a machine called Hinoki with the FPGA board Xilinx Alevo U50 attached on it. Hinoki has a x86_64 Architecture and the CPU is Intel(R) Xeon(R) Gold 6238R CPU @ 2.20GHz. The program is compiled using -O0 optimization option and with g++. The execution time is measured using the steady_clock of the "chrono" library from STL and default time command. The 4 benchmarks 3d-rendering, digital recognition, optical flow, spam-filter are from Rosetta benchmark[57], a realistic high-level synthesis benchmark suite for software programmable FPGAs.

6.2 Benchmarks

6.2.1 Rosetta

Rosetta is a comprehensive benchmark suite for software-programmable FPGAs, specifically designed to emulate real-world scenarios. The suite encompasses fully-developed applications from machine learning and image/video processing domains. Each benchmark comprises multiple compute kernels that reveal various sources of parallelism. These applications have been developed considering realistic design constraints and are optimized at both the kernel and application levels using the sophisticated features of high-level synthesis (HLS) tools to adhere to these constraints.

6.2.2 Vitis Accel Examples

Vitis Accel Examples is developed by Xilinx, serve not only as a comprehensive resource for software developers working with the Vitis Unified Software Platform but also as a benchmark for evaluating the performance of FPGA-based hardware acceleration

solutions. By offering a diverse set of examples and use cases, Vitis Accel Examples enables developers to assess the effectiveness of their custom designs against reference implementations in various domains such as machine learning, image processing, and data analytics.

6.3 System Overhead

In this section, we offer a comprehensive assessment of the proposed framework, drawing upon experimental results. We have evaluated four distinct metrics: native execution time, function execution time, Urunc deploy time, and watchdog initialization time. A detailed examination of these metrics allows us to gauge the advantages and disadvantages of the framework in question.

- The native execution time serves as a benchmark for performance comparisons, representing the time needed to execute the application directly through the Funky monitor. This metric provides a baseline against which other deployment methods, such as Urunc, can be assessed. To measure the native execution time, a steady_clock is initiated at the beginning of the function, and the duration is recorded upon the function's completion. To incorporate the reconfiguration time into the evaluation, it is necessary to reconfigure the FPGA (Field-Programmable Gate Array) board with different logic each time after deploying the benchmark application. This additional step ensures that the performance comparison accounts for the time required to reconfigure the FPGA, thus providing a more comprehensive understanding of the system's overall efficiency and performance.
- The function execution time covers the entire period from submitting an HTTP request to the function's watchdog until a response is received. To measure the function time, we first deploy the function using urunc on our host, and then send an HTTP request to the watchdog via *curl*. The *time* command is used to track the time needed to execute the curl command. Again, the reconfiguration time should be included in this measurement.
- The Urunc deploy time metric reveals the time necessary to deploy the serverless function with Urunc, shedding light on the overhead resulting from the containerization process. This metric is determined by analyzing the log data of urunc and calculating the duration from the initiation of the agent until the actual execution of the unikernel deployment command.
- The watchdog initialization time evaluates the time needed for the watchdog to establish the listening port and register the event in the event pool. The method

employed to measure this time is identical to the first metric.

We executed the function five times for each benchmark to measure all four metrics and subsequently calculated the average values, which are presented in the table shown below.

6.3.1 Overhead Analysis

Benchmarks	**Size (KB)**
3d-rendering	34904
Digital Recognition	37993
Optical Flow	62919
Spam-Filter	117819

Table 6.1: Comparison of binary size of different benchmarks

The deploy time, as shown in Figure 6.1, measures the time elapsed from the moment the Urunc agent is initiated until the command to execute the Funky Monitor is issued by Urunc. This metric is crucial because it reflects the real overhead introduced by Urunc, excluding the time consumed by the orchestrator and the pre-processing by Kata container. This evaluation allows for a more accurate understanding of the system's efficiency in deploying serverless functions.

Upon examining Figure 6.1, it becomes apparent that the deploy time varies across different benchmarks. This variability can be attributed to the differing data sizes of the benchmarks, as Urunc first extracts the unikernel binary from the image before executing it. This conclusion is further supported by the data presented in Table 6.1, which confirms the correlation between data size and deploy time.

Meanwhile, Figure 6.2 illustrates the watchdog initialization time, which encompasses the time required to create and listen to a port for incoming messages, as well as the time needed to register the event for handling these messages. Figure 6.2 reveals that the watchdog initialization time remains relatively constant across all benchmarks and is negligible when compared to the deploy time previously discussed.

6.3.2 Performance Analysis

In this section, we compare the performance of our newly proposed system and the original Funky monitor. This comparison aims to validate whether the Urunc-deployed applications maintain the same performance level as the native Funky application.

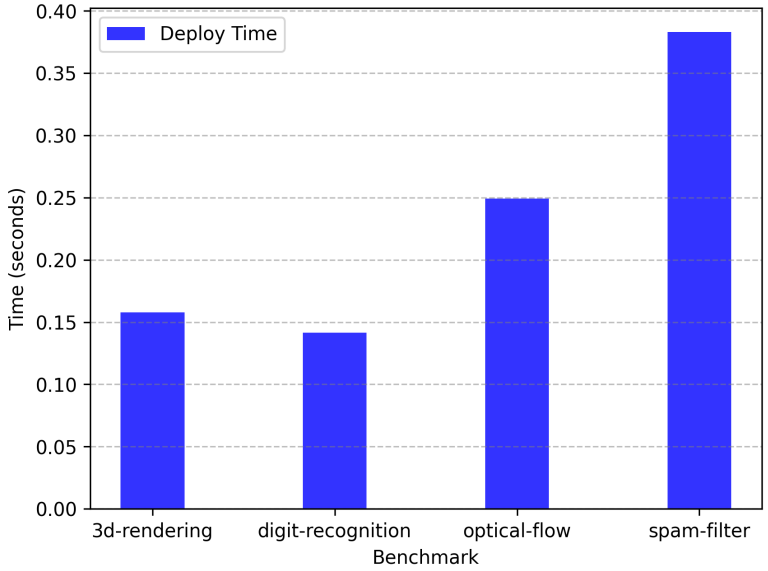

Figure 6.1: deploy time

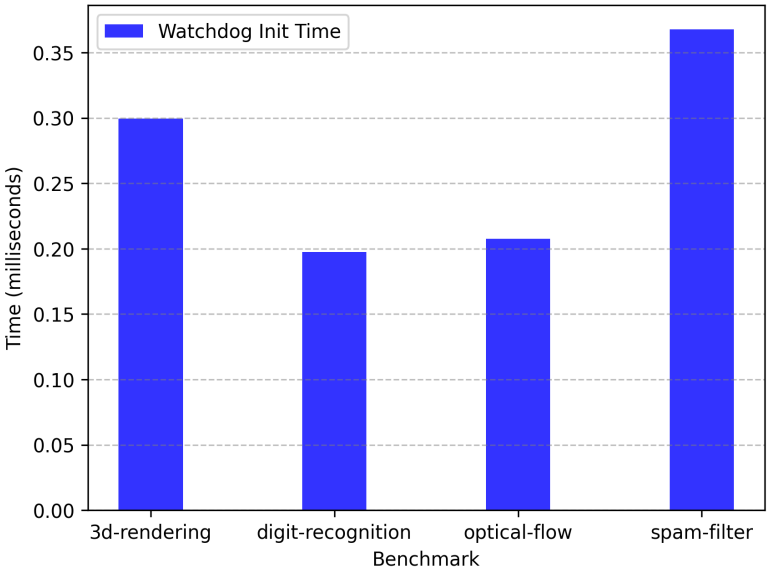

Figure 6.2: watchdog initialize time

As mentioned earlier, Urunc deploys the unikernel directly on the host, leading to the expectation that Urunc-deployed applications should exhibit similar performance characteristics to the original Funky-deployed applications. To validate this hypothesis, a comparison of the function time of Urunc-deployed applications and the execution time of native Funky-deployed applications is presented in Figure 6.3 and Figure 6.4.

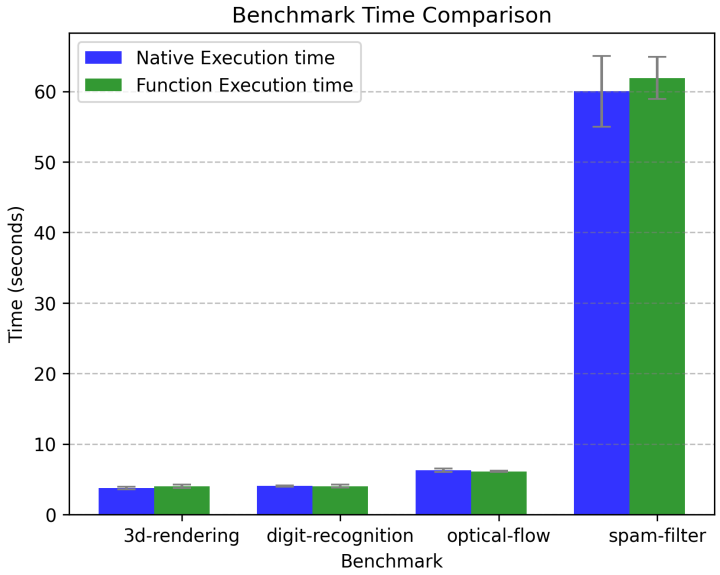

Figure 6.3: Compare the performance of Native Funky deployed application and urunc deployed Rosetta applications

Upon examining Figure 6.3 and Figure 6.4, it is evident that there is no significant difference in execution time between the Urunc-deployed and Funky-deployed applications, as initially expected. The overall performance differences of Rosetta is 1.89% faster and 1.77% faster for Vitis Accel Examples. This result can be attributed to the fact that the tests were conducted locally, with input data being read directly from local files rather than being transferred over a network. Otherwise, the network latency will also be shown in the figure. Consequently, this evaluation demonstrates that the newly proposed system performs on par with the original Funky monitor, maintaining comparable execution times and validating the system's effectiveness in deploying serverless functions.

In conclusion, the performance assessment of Urunc validates its efficiency and competitiveness with the original Funky monitor. By demonstrating minimal time differences in the function execution of the applications, this evaluation substantiates the claim that Urunc is capable of maintaining the same level of performance as its

native Funky counterpart, thus proving its suitability as an alternative system for deploying serverless functions.

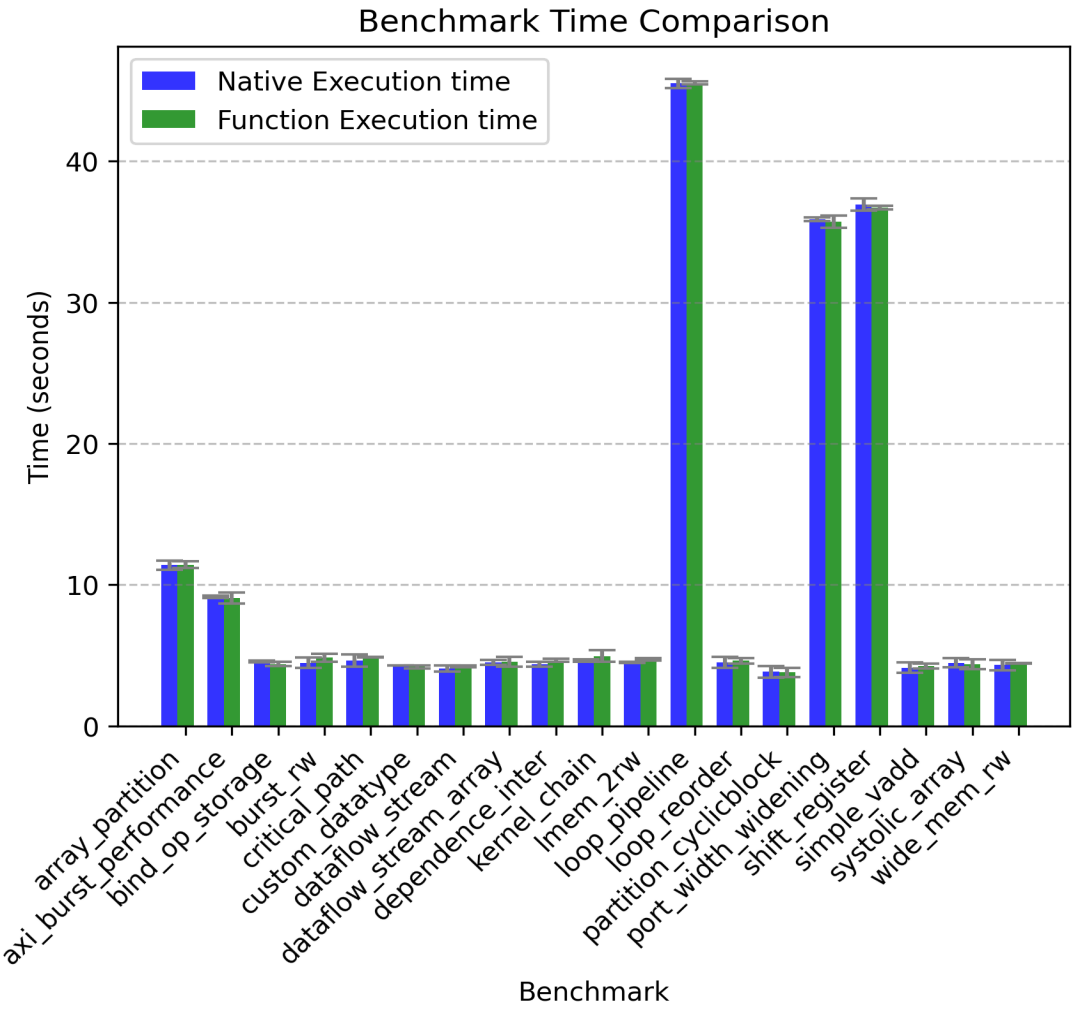

Figure 6.4: Compare the performance of Native Funky deployed application and urunc deployed Vitis Accel Examples applications

7 Related Work

7.1 Unikernels

Unikernel architecture offers lightweight virtualization through the use of specialized library operating systems that are linked directly to a single application. These unikernel-based systems consist of small, single-address-space images that incorporate the application, its configuration, and any application-dependent OS components. This approach aims to enhance security, performance, and resource efficiency in virtualization. Various frameworks have been developed to address different unikernel objectives, including Unikraft[34] for specialized, high-performance unikernels, IncludeOS[9] for minimal, resource-efficient unikernels, and HermitCore[37] for extreme-scale computing.

7.1.1 Unikraft

Unikraft[34] is a novel micro-library operating system aimed at facilitating the generation of specialized, high-performance unikernels. It addresses the issues of significant expert work needed for building unikernels and their non-POSIX compliance by providing a fully modular kernel and performance-oriented, composable APIs. The Unikraft kernel enables customization, while its APIs offer easy selection and composition of core OS components, resulting in easily replaceable and selectable components based on performance needs.

Unikraft has ported the musl libc library and provides a syscall shim layer micro-library, which simplifies the process of running an application on Unikraft. The platform supports numerous applications, programming languages, runtime environments, and hypervisors/VMMs. The evaluation shows that Unikraft delivers a 1.7x-2.7x performance improvement compared to Linux guests while consuming minimal resources, and its images boast fast boot times.

7.1.2 IncludeOS

IncludeOS[9] is a single-tasking library operating system designed for cloud services, offering highly specialized components and minimal resource overhead compared to

classical general-purpose operating systems. Written in C++, IncludeOS is tailored for virtualized environments and provides extreme resource efficiency, a small disk and memory footprint, efficient deployment, and virtualization platform independence.

IncludeOS allows developers to build C++ code directly into a virtual machine at compile-time, eliminating unnecessary overhead and improving performance. The system is highly modular, with only required parts included, and uses an event-based asynchronous I/O model. The resulting disk image is compatible with various virtualization environments, including OpenStack and VirtualBox.

The benefits of IncludeOS over traditional Linux virtual machines include significantly smaller disk and memory footprints, increased overall performance, virtual x86 hardware support with standard virtio networking, fast boot times (less than 0.3 seconds), no system call overhead, and a reduced number of VM exits due to a low count of protected instructions.

7.1.3 HermitCore

As high-performance computing (HPC) hardware becomes increasingly complex, the development of the entire software stack from operating system (OS) to application layer is burdened. The HPC community is driven to create larger and more efficient supercomputers using the latest hardware trends, which results in additional challenges for software development. The operating system must be flexible, adaptable, and scalable in response to these changes. To address these requirements, a new unikernel operating system design for HPC called HermitCore[37] is invented.

HermitCore extends the multi-kernel approach by combining the advantages of a unikernel and a multi-kernel design, offering a single-address-space kernel that promises low overhead and excellent scalability. By using Linux on a small subset of cores as service nodes, HermitCore allows for backward compatibility while focusing on mapping the hardware to the software structure. The unikernel is designed to reduce OS noise, ensure predictable runtimes, maintain flexibility and extensibility, and abstract hardware details, while supporting common HPC programming models such as OpenMP and MPI.

7.1.4 Funky Unikernel

Funky unikernel is based on IncludeOS, which can be categorized as a unikernel that straddles the line between clean-state and POSIX-like unikernel families. Built from the ground up using C++, IncludeOS enables developers to convert their C/C++ applications into unikernels using a custom GCC toolchain. This makes it an ideal choice for implementing OpenCL applications since OpenCL host APIs are designed primarily for

C/C++ code. By leveraging the strengths of IncludeOS, funky unikernel aim to create a unikernel that provides a suitable environment for high-performance computing (HPC) applications while maintaining compatibility with established programming models and integrating seamlessly with existing HPC software stacks.

7.2 FPGA in cloud environments

We have organized the literature into three main categories: communication techniques, sharing strategies, and computational paradigms.

7.2.1 Communication Techniques

PCIe-Passthrough

Firstly, communication techniques pertain to the methodologies utilized for accessing shared or virtualized devices. The most fundamental method, PCIe-Passthrough, establishes a direct connection between a Virtual Machine (VM) or container and the FPGA device, exemplified by instances like AWS F1. Paravirtualization entails connecting the requesting application VM to a host device driver, which subsequently virtualizes resource access, as illustrated in the case of pvFPGA [50]. This paper investigates FPGA virtualization for application as a hardware accelerator in cloud computing environments using the Xen Virtual Machine Monitor (VMM). The authors concentrate on developing a mechanism enabling user processes to access the FPGA accelerator with minimal overhead, facilitating simultaneous requests from multiple guest domains for shared FPGA accelerator access, and allowing special domains (VIP domains) to complete requests faster than ordinary domains when contending for access. The proposed solution, called pvFPGA, comprises an FPGA hardware accelerator design and FPGA virtualization on an x86 server. The hardware accelerator implementation utilizes the PCIe interface and Direct Memory Access (DMA) technique for efficient data transfer. The FPGA virtualization design incorporates frontend and backend drivers, along with a genuine device driver for the FPGA accelerator. PvFPGA employs a shared-memory mechanism for low overhead data transfer between domains.

API Remoting

Among the analyzed methods, API Remoting is the most prevalent[25, 4, 40, 58], employing a custom API for remote device access and facilitating multiple applications to manage the shared device via space and time sharing. BlastFunction[18] serves as an example, constituting an FPGA sharing system designed to accelerate microservices

and serverless applications in cloud environments. It allows concurrent kernel execution on the same FPGA for multiple applications without altering the underlying host code. This is achieved through leveraging OpenCL as the accelerators' runtime support system. BlastFunction allocates available devices based on application requests and utilizes runtime metrics collected by the system. The primary components of BlastFunction include the Remote OpenCL Library, the Device Manager, and the Accelerators Registry. The Remote OpenCL Library enables client applications to integrate with BlastFunction through a custom OpenCL implementation, abstracting the remote device access protocol and communication with the Accelerators Registry and Device Managers. Each Device Manager connects to an FPGA in the system and provides a time-sharing mechanism, granting isolated access for multiple application containers. The Accelerators Registry functions as the central controller, managing allocation and reconfiguration of available devices using runtime performance metrics. BlastFunction integrates with external components like the Cloud Orchestrator (Kubernetes) and the Gateway (OpenFaaS) for controlling cluster resources and handling autoscaling. The system supports both synchronous/blocking OpenCL calls and asynchronous/non-blocking calls to the remote runtime, ensuring application consistency by executing command-queue methods in the correct order.

Both BlastFunction and our system exhibit similarities in their underlying structure, as they utilize Kubernetes and OpenFaaS as the frontend. However, significant disparities emerge in application deployment, resource utilization, and optimization potential. Our system incorporates the Funky Unikernel for application deployment, presenting a more lightweight and secure solution compared to BlastFunction. Unikernels are specialized, single-address-space machine images constructed by compiling an application with only the necessary parts of the operating system. By eliminating superfluous components, the Funky Unikernel reduces memory footprint and attack surface of applications, resulting in enhanced performance and security. One of our system's key features is the migration function provided by the Funky Monitor. This functionality enables higher utilization of Field Programmable Gate Array (FPGA) boards across different nodes, and creates the potential for deploying distinct functions with identical kernel functions on the same board. Consequently, this approach can conserve time spent on FPGA board reconfiguration, ultimately reducing cold boot time. In summary, although BlastFunction and our system share frontend structure similarities, our system holds several advantages in terms of application deployment and resource utilization. By leveraging Funky Unikernel and the migration function of Funky Monitor, our system demonstrates enhanced security, performance, and optimization potential.

A variant of API Remoting is depicted in [43], which exposes a microservice for individual accelerators instead of a general API for the entire system. The FaaM design and implementation is based on Docker containers and can be applied to other

virtualization technologies and FPGA platforms. It consists of a Worker Node and a Service Node. The Worker Node hosts one or more Worker Containers, which abstract specific hardware accelerator functions, while the Service Node hosts FaaM services, providing hardware-accelerated functions. To bridge the gap between native C/C++ code and Java-based applications, Java Native Interface (JNI) and SWIG are used to generate Java wrappers from native C++ classes. The Java NIO mechanism is utilized to efficiently handle buffer movement between Java's heap memory and native memory used by the FPGA accelerator. The implemented task scheduler uses a first-come-first-serve (FCFS) policy to maintain fair sharing of the FPGA accelerator.

Direct Network Access

Lastly, Direct Network Access, employed by solutions such as Catapult [13], offers low-latency access by making the FPGA available through its network interface.

7.2.2 Sharing Strategies

Space Sharing

Sharing strategies focus on the ways devices are shared among applications. Space-Sharing [11, 4, 59] leverages FPGA virtualization using Partial Dynamic Reconfiguration (PDR) or Overlays, enabling the simultaneous execution of multiple accelerators on the same FPGA for different applications. Although space-sharing allows for the complete utilization of device resources, it demands careful management of accelerators to minimize reconfiguration times.

FPGAPooling manages FPGA accelerators as a single resource pool shared among all VMs, allowing VMs to request an FPGA accelerator from the pool when needed, and releasing it back to the pool when finished.

The FPGAPooling system in [59] is designed with a centralized scheduler that handles acceleration requests from VMs, assigning them to idle FPGA accelerators at runtime. The system provides abstraction of FPGA resources, easy and efficient access to FPGA acceleration services, and efficient use of valuable FPGA resources. Multiple scheduling algorithms are designed and implemented for the FPGAPooling system to handle multi-dimensional resource constraints and different workload distributions.

The proposed FPGAPooling system offers abstraction, heterogeneity, easy accessibility, and robust performance, overcoming limitations of existing approaches and providing a more efficient and flexible solution for FPGA resource management and sharing in cloud environments.

The authors in [4] propose a methodology that simplifies FPGA application development by providing clean abstractions with high-level APIs and a flexible execution

model that supports both software and hardware execution. To enhance device utilization and enable sharing of FPGAs among multiple users, a lightweight runtime system featuring hardware-assisted memory virtualization and memory protection is introduced, allowing concurrent execution of multiple applications.

The proposed system architecture connects a host computer in a cloud environment to an FPGA board via a PCIe interface and employs partial reconfiguration for the dynamic instantiation of hardware accelerators. The runtime manager, built on top of a modified FreeRTOS kernel, is responsible for FPGA resource management and communication with the host. The virtualization infrastructure mediates memory accesses, implementing memory protection and address translation.

The design flow allows users to generate applications from high-level domain-specific language (DSL) specifications or by providing high-level synthesis (HLS) or RTL specifications. The toolchain automatically generates an FPGA application package containing the partial bitstreams and code for the onboard processor.

Time Sharing

Time-Sharing[50, 25, 40, 43, 13] addresses the challenge mentioned above by multiplexing requests from multiple applications on a single accelerator within an FPGA board. The primary challenge lies in efficiently scheduling incoming requests to reduce latency and managing memory access within the constraints of I/O bandwidth.

In [13], the authors introduce an innovative cloud architecture, termed Configurable Cloud, which employs reconfigurable logic (FPGAs) to expedite network plane functions and applications within hyperscale datacenters. This architectural design situates FPGAs amidst network switches and servers, facilitating programmable transformations of network flows, acceleration of local applications, and direct inter-FPGA communication on a datacenter scale.

Configurable Cloud addresses the limitations encountered by preceding designs, such as intricate cabling systems, failure management, and restricted direct communication between FPGAs. This adaptable and scalable architecture enables FPGAs to serve as both network accelerators and local computational accelerators while fostering direct inter-FPGA communication across the datacenter. The authors introduce a Lightweight Transport Layer (LTL) protocol, which supports low-latency connections between FPGAs, and demonstrate the efficacy of this architecture in various contexts, such as web search and network flow encryption scenarios.

7.2.3 Computational Paradigms

Computational paradigms are differentiated by how workloads and FPGA connections are managed. Batch Systems[50, 4, 59, 11] treat workloads as time-limited entities, with the scheduling and allocation of workloads determined by each job's duration. In contrast, Service-Based systems[25, 40, 43, 13] maintain continuous FPGA operation by processing incoming requests from the system or applications. Some works, such as [43], expose the FPGA accelerator as a standalone service.

In the study conducted by [40], the researchers present VineTalk, a novel framework aimed at diminishing the programming complexities associated with FPGA-based accelerators and virtualization in data centers. FPGA-based accelerators demonstrate enhanced energy efficiency and computational densities for workloads with a high sensitivity to latency. Nevertheless, they pose certain challenges for both application developers and cloud providers, such as intricate interfacing and an absence of sharing mechanisms.

To tackle these issues, VineTalk is devised to establish a software layer between FPGAs and applications, streamlining the interaction between application software and accelerator hardware while facilitating the sharing of FPGA resources among various applications. The framework is compatible with native server applications, virtual machines, and containers, and encompasses two main components: a Communication Layer, which implements virtual accelerators, and a Software Controller responsible for scheduling and managing access to the accelerators.

VineTalk demonstrates a promising methodology for the transparent utilization of FPGAs in data center servers by alleviating programmer burden and offering low-overhead access to accelerators as well as sharing capabilities.

8 summary and Conclusion

In this master thesis, we have successfully integrated existing resources, including OpenFaaS, Kubernetes, vAccel, Urunc, and Funky Monitor, to build a serverless framework for unikernel applications that can leverage FPGA boards for acceleration. We have achieved several important milestones in the process:

8.1 Watchdog Implementation for Funky Unikernel Applications

We implemented a watchdog for Funky unikernel applications, enabling event-driven functionality in the framework. This enhancement allows for more efficient resource utilization by only triggering functions when specific events occur.

8.2 Event Loop Extension in Funky Unikernels

We extended Funky unikernels to ensure that functions continue running after the main function returns if there are registered events in the event loop. This modification improves the responsiveness and adaptability of the system in dynamic environments.

8.3 FPGA Programming API Design

We proposed a new design for an FPGA programming API that simplifies the usage of FPGA boards and encourages more users to implement their own FPGA-accelerated functions. This design aims to make FPGA acceleration more accessible and appealing to a wider range of developers.

8.4 Framework Evaluation

We evaluated the new framework and demonstrated that its performance is on par with the original Funky Monitor deployed application, except for the deployment overhead and network latency. The deployment overhead is introduced by Urunc,

while network latency is due to the watchdog. This evaluation highlights the potential of the framework in delivering efficient and high-performance unikernel applications with FPGA acceleration.

In conclusion, the work presented in this thesis has laid the foundation for a serverless framework that enables unikernel applications to efficiently utilize FPGA acceleration. Further development and enhancements, as outlined in the Future Work section, will ensure that the framework becomes even more versatile, accessible, and performant in the future.

9 Future Work

In this section, we outline several future directions for the development and improvement of our work presented in this master thesis. These directions are aimed at enhancing the overall functionality and ease of use of the system.

9.1 Watchdog Improvement and Automation

The current watchdog implementation can be improved by automating the process of compiling developer-defined functions alongside the watchdog itself. This automation would streamline the development process and reduce the potential for human errors during compilation. The support for concurrent requests and health checks should also be implemented, to ensure that event-driven execution is reliable and efficient.

9.2 User-friendly Interface for FPGA Acceleration

We plan to implement the interface designed in this thesis to simplify the process of using FPGA boards for accelerating user-defined functions. By providing an accessible and intuitive interface, developers can more efficiently leverage FPGA acceleration in their projects.

9.3 Flexible Developer Inputs in urunc

The functionality of urunc should be extended to receive more flexible developer inputs through the command line interface. This enhancement will provide developers with greater control and customization options when using urunc in their projects.

9.4 Event Loop Extension for Selective Looping

The event loop should be modified to only loop when necessary, such as in the case of the watchdog that requires continuous monitoring for external requests. In situations where the process does not need to run in the background for an extended period, the event loop should be able to terminate gracefully.

9.5 FPGA-aware Kubernetes Scheduler

Adapting the Kubernetes scheduler to be FPGA-aware would enable it to make intelligent decisions regarding function scheduling based on available FPGA resources and the specific kernel configurations on the FPGA board. This enhancement would lead to more efficient utilization of FPGA resources and improved overall system performance.

Abbreviations

List of Figures

List of Tables

Bibliography

[1] E. Adams and S. Desnoyers. "Kata containers: One year later." In: (2019).

[2] Amazon. *Amazon EC2 F1 instances*. 2021. URL: https://aws.amazon.com/ec2/instance-types/f1.

[3] E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, E. Komp, R. Sass, and D. Andrews. "Enabling a Uniform Programming Model Across the Software/Hardware Boundary." In: *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. 2006, pp. 89–98. DOI: 10.1109/FCCM.2006.40.

[4] M. Asiatici, N. George, K. Vipin, S. A. Fahmy, and P. Ienne. "Designing a virtual runtime for FPGA accelerators in the cloud." In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2016, pp. 1–4.

[5] O. G. Attia, T. Johnson, K. Townsend, P. Jones, and J. Zambreno. "Cygraph: A reconfigurable architecture for parallel breadth-first search." In: *2014 IEEE International Parallel Distributed Processing Symposium Workshops*. 2014, pp. 228–235. DOI: 10.1109/IPDPSW.2014.131.

[6] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah. "Lime: A Java-Compatible and Synthesizable Language for Heterogeneous Architectures." In: *SIGPLAN Not.* 45.10 (Oct. 2010), pp. 89–108. ISSN: 0362-1340. DOI: 10.1145/1932682.1869469. URL: https://doi.org/10.1145/1932682.1869469.

[7] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. "Chisel: Constructing hardware in a scala embedded language." In: *DAC Design Automation Conference 2012*. 2012, pp. 1212–1221.

[8] J. Bhasker. *A vhdl primer*. Prentice-Hall, 2021.

[9] A. Bratterud, A.-A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum. "Includeos: A minimal, resource efficient unikernel for cloud services." In: *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2015, pp. 250–257.

[10] B. Burns, J. Beda, and K. Hightower. "Kubernetes: Up and running: Dive into the future of infrastructure." In: *O'Reilly Media, Inc.* 2016.

[11] S. Byma, J. G. Steffan, H. Bannazadeh, A. Leon-Garcia, and P. Chow. "FPGAs in the cloud: Booting virtualized hardware accelerators with OpenStack." In: *Proceedings - 2014 IEEE 22nd International Symposium on Field-Programmable Custom Computing Machines, FCCM 2014*. 2014, pp. 109–116. DOI: 10.1109/FCCM.2014.39.

[12] J. Casper and K. Olukotun. "Hardware acceleration of database operations." In: *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. New York, NY, USA, 2014, pp. 151–160. DOI: 10.1145/2541940.2541944.

[13] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. "A cloud-scale acceleration architecture." In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016, pp. 1–13. DOI: 10.1109/MICRO.2016.7783729.

[14] E. S. Chung, J. D. Davis, and J. Lee. "Linqits: Big data on little clients." In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ACM. New York, NY, USA, 2013, pp. 261–272. DOI: 10.1145/2485922.2485944.

[15] A. Cloud. *Alibaba cloud fpga instances*. 2021. URL: https://www.alibabacloud.com/help/en/doc-detail/108504.html.

[16] V. Cozzolino, O. Flum, A. Y. Ding, and J. Ott. "Miragemanager: Enabling stateful migration for unikernels." In: *Proceedings of the Workshop on Cloud Continuum Services for Smart IoT Systems*. ACM. 2020, pp. 13–19.

[17] G. Dai, Y. Chi, Y. Wang, and H. Yang. "FPGP: Graph processing framework on FPGA–a case study of breadth-first search." In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. New York, NY, USA, 2016, pp. 105–110. DOI: 10.1145/2847263.2847285.

[18] A. Damiani, G. Fiscaletti, M. Bacis, R. Brondolin, and M. D. Santambrogio. "BlastFunction: A Full-Stack Framework Bringing FPGA Hardware Acceleration to Cloud-Native Applications." In: *ACM Trans. Reconfigurable Technol. Syst.* 15.2 (Jan. 2022). ISSN: 1936-7406. DOI: 10.1145/3472958. URL: https://doi.org/10.1145/3472958.

[19] P. M. Donald Thomas. *The verilog hardware description language*. Springer Science & Business Media, 2021.

[20] A. Ellis. *Introducing OpenFaaS: Serverless Functions Made Simple*. 2017. URL: https://blog.alexellis.io/introducing-functions-as-a-service/.

[21] I. FPGA. *Accelerator Functional Unit Developer's Guide for Intel FPGA Programmable Acceleration Card*. 2021. URL: https://www.intel.com/content/www/us/en/programmable/documentation/bfr1522087299048.html.

[22] I. FPGA. *Accelerator functional unit developer's guide for intel fpga programmable acceleration card*. 2021. URL: https://www.intel.com/content/www/us/en/programmable/documentation/bfr1522087299048.html.

[23] I. FPGA. *Intel fpga sdk for opencl pro edition: Programming guide*. 2021. URL: https://www.intel.com/%20content/www/us/en/programmable/documentation/%20mwh1391807965224.html.

[24] K. Group. *The opencl specification*. 2021. URL: https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html.

[25] A. Iordache, G. Pierre, P. Sanders, J. G. de Farias Coutinho, and M. Stillwell. "High performance in the cloud with FPGA groups." In: *Proceedings of the 9th International Conference on Utility and Cloud Computing*. ACM. 2016, pp. 1–10.

[26] K. Kara and G. Alonso. "Fast and robust hashing for database operators." In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. 2016, pp. 1–4. DOI: 10.1109/FPL.2016.7577318.

[27] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach. "Sharing, protection, and compatibility for reconfigurable fabric with amorphos." In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association. Carlsbad, CA, Oct. 2018, pp. 107–127.

[28] Khronos Group. *The OpenCL Specification*. Accessed: March 20, 2023. 2021. URL: https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html.

[29] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov. "OSv—optimizing the operating system for virtual machines." In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association. 2014, pp. 61–72.

[30] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun. "Spatial: A Language and Compiler for Application Accelerators." In: 53.4 (2018). ISSN: 0362-1340. DOI: 10.1145/3296979.3192379. URL: https://doi.org/10.1145/3296979.3192379.

[31] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun. "Automatic generation of efficient accelerators for reconfigurable hardware." In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016, pp. 115–127. DOI: 10.1109/ISCA.2016.18.

[32] D. Korolija, T. Roscoe, and G. Alonso. "Do OS abstractions make sense on FPGAs?" In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association. 2020, pp. 991–1010.

[33] Kubernetes. *Container Runtime Interface (CRI)*. 2020. URL: https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/.

[34] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, Ş. Teodorescu, C. Răducanu, C. Banu, L. Mathy, R. Deaconescu, C. Raiciu, and F. Huici. "Unikraft: Fast, specialized unikernels the easy way." In: *Proceedings of the Sixteenth European Conference on Computer Systems*. ACM. 2021, pp. 376–394.

[35] D. Kwon, J. Boo, D. Kim, and J. Kim. "FVM: FPGA-assisted virtual device emulation for fast, scalable, and flexible storage virtualization." In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association. 2020, pp. 955–971.

[36] S. Lankes, S. Pickartz, and J. Breitbart. "HermitCore: A Unikernel for Extreme Scale Computing." In: *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*. ROSS '16. Kyoto, Japan: Association for Computing Machinery, 2016. ISBN: 9781450343879. DOI: 10.1145/2931088.2931093. URL: https://doi.org/10.1145/2931088.2931093.

[37] S. Lankes, S. Pickartz, and J. Breitbart. "Hermitcore: A unikernel for extreme scale computing." In: *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*. ACM. 2016, pp. 1–8.

[38] I. Lebedev, C. Fletcher, S. Cheng, J. Martin, A. Doupnik, D. Burke, M. Lin, and J. Wawrzynek. "Exploring many-core design templates for FPGAs and ASICs." In: *Int. J. Reconfig. Comput.* 2012 (Jan. 2012), pp. 1–15. DOI: 10.1155/2012/523285.

[39] S. Li, H. Lim, V. W. Lee, J. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, S. O, S. Lee, and P. Dubey. "Architecting to achieve a billion requests per second throughput on a single key-value store server platform." In: *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 2015, pp. 476–488. DOI: 10.1109/ISCA.2015.31.

[40] S. Mavridis, M. Pavlidakis, I. Stamoulias, C. Kozanitis, N. Chrysos, C. Kachris, D. Soudris, and A. Bilas. "VineTalk: Simplifying software access and sharing of FPGAs in datacenters." In: *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2017, pp. 2–5.

[41] J. M. Mbongue, D. T. Kwadjo, A. Shuping, and C. Bobda. "Deploying Multi-tenant FPGAs Within Linux-based Cloud Infrastructure." In: *ACM Trans. Reconfigurable Technol. Syst.* 15.2 (Dec. 2021). DOI: 10.1145/3471934.

[42] T. Oguntebi and K. Olukotun. "Graphops: A dataflow library for graph analytics acceleration." In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. New York, NY, USA, 2016, pp. 111–117. DOI: 10.1145/2847263.2847269.

[43] S. Ojika, A. Gordon-Ross, H. Lam, B. Patel, G. Kaul, and J. Strayer. "Using FPGAs as Microservices: Technology, Challenges and Case Study." In: *9th Workshop on Big Data Benchmarks Performance, Optimization and Emerging Hardware (BPOE-9)*. 2018.

[44] M. Paolino, S. Pinneterre, and D. Raho. "FPGA virtualization with accelerators overcommitment for network function virtualization." In: *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE. 2017, pp. 1–6. DOI: 10.1109/RECONFIG.2017.8279753.

[45] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. "A reconfigurable fabric for accelerating large-scale datacenter services." In: *IEEE Micro* 35.3 (2015), pp. 10–22. DOI: 10.1109/MM.2015.53.

[46] M. H. Quraishi, E. B. Tavakoli, and F. Ren. "A survey of system architectures and techniques for FPGA virtualization." In: (2021).

[47] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh. "From high-level deep neural models to FPGAs." In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016, pp. 1–12. DOI: 10.1109/MICRO.2016.7783747.

[48] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao. "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks." In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. New York, NY, USA, 2016, pp. 16–25. DOI: 10.1145/2847263.2847268.

[49] A. Vaishnav, K. D. Pham, and D. Koch. "A survey on FPGA virtualization." In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2018, pp. 131–1317. DOI: 10.1109/FPL.2018.00028.

[50] W. Wang, M. Bolic, and J. Parri. "PvFPGA: Accessing an FPGA-based hardware accelerator in a paravirtualized environment." In: *2013 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2013*. 2013.

[51] D. Williams and R. Koller. "Unikernel Monitors: Extending Minimalism Outside of the Box." In: *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'16. USA: USENIX Association, 2016, pp. 71–76.

[52] Xilinx. *XRT and Vitis Platform Overview*. 2021. URL: https://xilinx.github.io/XRT/master/html/platforms.html.

[53] Xilinx. *Xrt and vitis platform overview*. Last accessed 16 September 2017. 2021. URL: https://xilinx.github.io/XRT/master/html/platforms.html.

[54] H. Yu, A. M. Peters, A. Akshintala, and C. J. Rossbach. "Ava: Accelerated virtualization of accelerators." In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery. 2020, pp. 807–825. DOI: 10.1145/3373376.3378538.

[55] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. "Optimizing FPGA-based accelerator design for deep convolutional neural networks." In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. New York, NY, USA, 2015, pp. 161–170. DOI: 10.1145/2684746.2689086.

[56] Z. Zhao, H. Sadok, N. Atre, J. C. Hoe, V. Sekar, and J. Sherry. "Achieving 100 Gbps intrusion prevention on a single server." In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association. 2020, pp. 1083–1100.

[57] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. A. Velasquez, W. Wang, and Z. Zhang. "Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software-Programmable FPGAs." In: *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)* (Feb. 2018).

[58] Z. Zhu, A. X. Liu, F. Zhang, and F. Chen. "FPGA Resource Pooling in Cloud Computing." In: *IEEE Transactions on Cloud Computing* PP.c (2018), pp. 1–1. DOI: 10.1109/TCC.2018.2840093.

[59] Z. Zhu, A.-X. Liu, F. Zhang, and F. Chen. "FPGA Resource Pooling in Cloud Computing." In: *IEEE Transactions on Cloud Computing* PP.c (2018), p. 1.