SCHOOL OF MANAGEMENT

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Management & Technology

Cloud-Native Scheduling for Serverless FPGAs

Bruno Scheufler

SCHOOL OF MANAGEMENT

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Management & Technology

Cloud-Native Scheduling for Serverless FPGAs

Author:	Bruno Scheufler
Matriculation Number:	x
Supervising Chair:	Chair of Distributed Systems and Operating Systems
Supervisor:	Prof. Dr-Ing. Pramod Bhatotia
Advisor:	Charalampos Mainas, Dr. Atsushi Koshiba
Submission Date:	13.07.2023

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

Munich, 13.07.2023 Bruno Scheufler

Acknowledgments

First and foremost, I would like to thank my family and friends for their tremendous support and encouragement. Thanks to Hendrik, Nicolás, Jonas, and Tim for proofreading my thesis and providing valuable insights, as always.

I also thank my advisor Charalampos Mainas and my supervisor Prof. Dr.-Ing. Pramod Bhatotia for their valuable feedback and guidance throughout this project.

Finally, I would like to thank the chair of Distributed Systems & Operating Systems for giving me the opportunity to work on this project.

Abstract

Field-programmable gate arrays (FPGAs) are dynamically reconfigurable hardware devices that accelerate workloads by offloading compute-intensive tasks (encryption, compression, and machine learning) from the CPU, improving performance and energy efficiency while allowing for a high degree of customization at runtime. Hardware accelerators are increasingly used in the cloud, but the efficient use of FPGA resources in a multi-tenant environment remains a challenge, not least due to the prohibitively high reconfiguration times and scarce availability of FPGA resources.

Serverless computing platforms and Function-as-a-Service (FaaS) are popular deployment models for cloud applications, as they allow developers to focus on the application logic without having to worry about the underlying infrastructure. However, even when FPGAs are made accessible and applications can be synthesized to benefit from hardware acceleration, existing serverless platforms are not designed to support the large-scale deployment of FPGA-accelerated functions, lacking the necessary context of the underlying hardware and application requirements to make informed scheduling decisions. Key issues include managing the scarce availability of FPGA access and ensuring consistently balanced resource utilization.

In this project, we consider the impact of function placement on overall system performance, especially FPGA utilization and fairness. To this end, we propose *fpgascheduling*, an extension to the popular Kubernetes orchestration system, more specifically the Kubernetes scheduler, adding FPGA awareness based on metrics recorded at runtime, enabling the improved resource efficiency, fairness, and scalability of FPGA-accelerated workloads in a multi-tenant environment. For evaluation purposes, we also present a simulator based on production traces from a real-world serverless platform to evaluate the system performance in a large-scale deployment without requiring extensive hardware and system stack configuration.

Our experimental results show that the proposed scheduler extension achieves a more uniform distribution of workloads and increases the per-node utilization to make efficient use of the existing hardware. Most importantly, a more equal distribution of functions across nodes leads to higher fairness. Furthermore, we show that the proposed scheduler extension is compatible with existing Kubernetes deployments and can be used to add hardware acceleration support to existing serverless application platforms.

Contents

1 Introduction

Context: FPGAs are becoming increasingly popular for accelerating applications, offering high performance and energy efficiency compared to CPUs and GPUs. Especially for neural networks and deep learning tasks, FPGAs are a promising alternative to GPUs [61][63][71][16][68]. Other use cases for FPGAs include databases, which benefit from hardware acceleration [13][30][42][51], graph processing [4][18][47], storage virtualization [40], page search ranking [14][52], and image processing [2].

Serverless computing enables developers to focus on their application logic without worrying about the underlying infrastructure, while the cloud provider takes care of provisioning and scaling the infrastructure. FaaS is a form of serverless computing in which functions are executed in response to events. Functions are stateless and short-lived and are billed based on their execution time.

In the cloud, FPGAs have been made available in a fixed configuration [1], resulting in over-provisioned or under-provisioned compute resources. FPGAs are not accessible to many developers: Programming FPGAs is still a challenge and FPGAs are not supported out of the box by existing serverless computing platforms such as AWS Lambda [6], Google Cloud Functions [24], Azure Functions [7], OpenFaaS, and Knative [33]. Complicating matters further, bitstreams are not portable across different FPGAs or even different slots on the same FPGA, if split into multiple virtual FPGAs, which makes it difficult to share FPGAs between multiple users and to migrate applications between different FPGAs. Long reconfiguration times (the time required to load a new bitstream) in the order of seconds present a major bottleneck of using FPGAs in serverless computing, though recent work has been successful in reducing the reconfiguration time down to several milliseconds [35], using partial reconfiguration [55] and multi-FPGA systems [28][54]. Similarly, lots of research work has been done to reduce the cold start latency in serverless computing, such as pre-warming [60][32].

Major public cloud platforms offer a high level of flexibility for customers but experience low resource utilization between 10-40% across the board due to a high discrepancy between requested and consumed resources [59][17][49][20][21][10][53][27][22]. Underutilization of resources in large data centers leads to energy inefficiency, which has significant negative financial and environmental consequences [22][41][46], whereas overutilization can lead to performance degradation and even service failure.

Motivation: Enabling FPGAs for serverless computing can help accelerate appli-

cations, reduce the cost of running applications, increase the utilization of FPGAs in data centers, and make FPGAs accessible to more developers. Applications relying on heterogeneous computing use both the host CPU and the FPGA for acceleration tasks, so exclusively focusing on either the CPU or the FPGA is not sufficient for many production applications. While most related work focuses on the FPGA hardware implementation and optimizing cold starts, focusing on the workload scheduler may yield untapped upside for improving the performance of serverless applications. By adding FPGA awareness on the scheduler level, the system benefits from improved resource efficiency, increased system throughput, better scalability, more uniform workload distribution, improved reliability and resilience, higher fairness, and predictable performance. By scheduling pods on nodes with the lowest FPGA utilization, we are ensuring that FPGA resources are used optimally. This can reduce wastage and result in cost savings, particularly in large-scale deployments. If workloads are evenly distributed across nodes, each node is likely to process its share of workloads faster, leading to increased overall system throughput. With an extended scheduler, the system can potentially scale more efficiently by distributing new pods to underutilized nodes, thus avoiding overloading certain nodes while others are idle. If certain nodes are continually busy while others are under-utilized, this can create hotspots that might lead to performance degradation or even failures. By spreading the load, this risk is mitigated. An evenly distributed system is less likely to suffer from cascading failures, too. If one node experiences downtime, only a fraction of the overall workloads would be affected, and those can be more quickly and easily redistributed to the other nodes. Equally spreading workloads ensures that no single node is overloaded, thus maintaining a sense of fairness in resource utilization across nodes. Finally, in a well-balanced system, it is easier to predict the performance of new workloads based on the current system state. This proves useful for capacity planning and workload scheduling.

State of the art: Related work focuses on FPGAs in the cloud, cluster scheduling improvements, serverless scheduling and resource management, serverless workload characterization, FPGA architecture for multi-tenancy, and serverless FPGA sharing. FPGAs in the cloud enable developers to accelerate their application with dedicated hardware without having to buy and maintain the accelerators themselves [12][3][28][74][70]. Cluster scheduling work focuses on better allocation decisions, including dollar-cost optimization [15] and deadline-aware global allocation [67]. Serverless scheduling and resource management attempts to improve the performance of serverless applications by improving placement and orchestration of serverless functions, adding quality-of-service (QoS) awareness [66][26], using additional scheduling criteria based on monitoring data [69][50], incoming requests [64][65], package dependencies [5], and cluster-level data [29]. Other approaches to reduce cold starts include pre-baking functions [62], using queueing models [23], leveraging function composition [75], and

performing utilization-aware bin packing [25]. Multiple teams have attempted to characterize serverless workloads to understand real-world data to improve scheduling policies, resulting in predicting the next invocation [60], placing functions on unused compute capacity [72], and improving function density [73]. FPGA architecture for multi-tenancy is the foundation for serverless FaaS platforms to provide elastic hardware acceleration as a service. This includes partitioning FPGAs [55], using partial reconfiguration for faster bitstream loading [54], and adding OS abstractions [35]. Serverless FPGA sharing combines the architectural and scheduling efforts to provide serverless platforms with FPGA sharing. While this space is still nascent, there are some initial efforts to provide serverless FPGA sharing [54][35][9].

Research gap: While existing work has made significant progress in improving the performance of serverless computing, it does not cover the scheduling of workloads in a heterogeneous environment with hardware accelerators: Data provided by the serverless infrastructure are not leveraged for optimizing the resource utilization of FPGAs and fairness of the system. More complex heterogeneous systems require additional information such as bitstream size, onboard location, and FPGA vendor preferences to make optimal scheduling decisions. Existing work does not consider this information.

Problem statement: In this thesis, we address the following research questions to improve the performance of serverless computing with FPGAs:

- How can we schedule workloads aware of FPGAs in serverless computing?
- How can we increase the utilization of FPGAs in serverless computing?
- How can we reduce FPGA reconfigurations for serverless workloads?
- How can we increase fairness by improving placement decisions in serverless computing?

High-level approach and design: We propose *fpgascheduling*, an extension to the Kubernetes scheduler to add FPGA awareness to the Kubernetes orchestration system and a metrics collector for FPGA-related metrics. To evaluate the large-scale scheduling of serverless workloads, we propose a simulator based on production traces from a real-world serverless platform.

Implementation overview: We implemented the proposed *fpgascheduling* extension to the Kubernetes scheduler as well as the metrics collector. We integrated the metrics collector with Funky Monitor, a unikernel hypervisor with multi-tenant FPGA support. Lastly, we implemented the proposed simulator based on production traces from Microsoft Azure [60].

Evaluation overview: We evaluate the proposed extension to the Kubernetes scheduler to increase the utilization of FPGAs and reduce FPGA reconfigurations. To that

end, we use the proposed simulator to evaluate the large-scale FPGA-aware scheduling of serverless workloads. As, to our best knowledge, no prior work exists in the space of scheduling serverless functions with FPGA support, we compare our results to the baseline of not using FPGAs for workload scheduling, measuring the impact on resource utilization and fairness.

Impact: With this project, we aim to create a first step towards a serverless FPGA platform. We believe that this project will enable platform operators to provide serverless computing with FPGA support. This will enable developers to accelerate their applications with hardware without having to buy and maintain the hardware themselves. Furthermore, increasing the resource utilization of FPGAs will reduce the cost of running applications.

Contributions:

- We present *fpgascheduling*, an extension to the Kubernetes scheduler to add FPGA awareness built using the Kubernetes scheduler framework, allowing for easy integration into existing Kubernetes deployments
- We present a metrics collector for FPGA-related metrics used by the scheduler plugin, deeply integrated into Funky Monitor
- We present a simulator to evaluate the large-scale FPGA-aware scheduling of serverless workloads based on production traces [72]

2 Background

2.1 Serverless and FaaS

Serverless computing is a cloud computing model in which the cloud provider manages the underlying infrastructure. FaaS is a popular deployment model for serverless computing. In FaaS, applications are split into small, stateless functions that are executed on demand. This level of abstraction means developers do not need to worry about provisioning and scaling, instead, they upload their code and the cloud provider takes care of building, running, and scaling the application. Functions can be invoked by incoming HTTP requests or events from other services.

The first function invocation usually requires a *cold start* in case the underlying application must be provisioned and launched. The cold start latency is the time between the invocation and the first response. Cold starts are caused by the need to provision and scale the application. Subsequent invocations are faster as the application is already running and can be reused (the function is *warm*). The cold start latency is a critical metric for serverless platforms as it directly impacts the user experience.

In serverless computing, multiple functions supplied by different customers can be executed on the same physical machine, in what is called a multi-tenant system. This allows for better utilization of physical resources and reduces the cost of running applications. However, it also introduces the risk of noisy neighbor effects, tenants that impact the performance of other functions running on the same machine by consuming a disproportionate amount of resources. Noisy neighbor effects can lead to unpredictable performance and higher costs for tenants.

Functions are automatically scaled up and down based on the number of incoming requests. This allows dealing with dynamically evolving invocation patterns elastically, without having to provision for peak loads. Pre-warming is a technique to reduce cold start latency. It involves executing a function before it is invoked to reduce the cold start latency. Pre-warming can be implemented by executing a function periodically to prevent cold starts altogether or provisioning it in anticipation of an upcoming invocation.

The primary goals in serverless platforms are to reduce cold start latency and offer a high level of scalability while providing an ergonomic developer experience. The main challenges are to efficiently utilize the underlying hardware and to ensure a high level

of fairness in a multi-tenant environment.

2.2 FPGA

FPGAs are used to accelerate compute-intensive workloads, such as machine learning, compression, encryption, and other algorithms that can be parallelized and benefit from the high throughput and low latency of FPGAs. However, FPGAs are not suitable for all workloads and are not typically used for general-purpose computing. Applications supporting hardware acceleration are traditionally split into two parts: Code running on the host (CPU) and code running on the FPGA. The code running on the host is responsible for accepting requests, performing business logic, and communicating with the FPGA (driver).

The code running on the FPGA (*bitstream*) is responsible for accelerating specific parts of the application. The bitstream is loaded onto the FPGA and the driver is executed on the CPU. The driver communicates with the bitstream via PCIe and is responsible for configuring the bitstream and transferring data to and from the bitstream.

FPGAs are programmed using hardware description languages (HDLs), such as Verilog and VHDL, and high-level synthesis (HLS) tools, such as Vivado HLS and Intel HLS Compiler. HLS tools allow developers to write their application in a high-level language, such as C, C++, or OpenCL, and then generate the bitstream. However, the generated bitstream is not portable across different FPGAs and requires recompilation for each FPGA, shell design and version, and partial reconfiguration region.

Reconfiguration time is the time required to load a new bitstream onto the FPGA. Reconfiguration time is in the order of seconds depending on the bitstream size and shell. It is a major bottleneck for using FPGAs in serverless computing, making up the majority of a cold start. Reconfiguration time can be reduced by using techniques such as pre-warming and partial reconfiguration.

Partial reconfiguration is the ability to load a new bitstream onto a part of the FPGA while the rest of the FPGA is still running. Partial reconfiguration allows for reducing the reconfiguration time down to several milliseconds [35] by only reconfiguring the part of the FPGA that needs to be reconfigured. However, partial reconfiguration is not supported by all FPGAs and requires additional hardware resources, such as partial reconfiguration controllers.

To enable multi-tenancy by design, the FPGA is split into a static shell region and multiple dynamic slots with partial reconfiguration support. Each of these slots can house a user-provided function bitstream. When a bitstream required for handling a function invocation is already present on a device (*bitstream locality*), no reconfiguration is required. We use bitstream locality alongside FPGA utilization to inform the

scheduler about function placement impact.

The main challenges in using FPGAs are the long reconfiguration time and the lack of portability of bitstreams across FPGAs manufactured by different vendors, running varying shell designs, and using different partial reconfiguration controllers.

2.3 Funky

This project makes use of Funky, a solution for running FPGA-enabled serverless workloads proposed by the Chair of Distributed Systems & Operating Systems at TUM[1]. With Funky, workloads are run in unikernels, communicating with available FPGAs through the hypervisor layer. Unikernels are specialized, lightweight, single-address-space machine images constructed by using library operating systems [43]. Including only necessary libraries, unikernels are considered more secure than traditional operating systems due to the smaller attack surface.

Funky uses IncludeOS [11], a popular library operating system written in C++, providing a modular interface for building applications and acting as the unikernel foundation. OpenCL [48] is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors. Funky extends OpenCL with hypervisor integration for shared FPGA usage. Xilinx Runtime (XRT) is a runtime library for Xilinx FPGAs, providing an OpenCL extension for programming FPGAs, and is used by Funky to communicate with the FPGA.

2.4 Cluster Scheduling and Load Balancing

Orchestration systems are responsible for managing the lifecycle of containers. They are used to manage containerized applications and provide features such as automatic scaling, load balancing, and service discovery. Given a set of configurations, they are responsible for creating the desired state of the system. Orchestration systems are also responsible for monitoring the system and making sure that the desired state is maintained in a control loop.

Kubernetes [37] is an industry-standard open-source container orchestration system. Containerized applications are represented as pods, which are groups of one or more containers that are deployed together on the same host. Pods are scheduled onto nodes, which are physical or virtual machines. Kubernetes uses a declarative approach to manage the state of the system. Users specify the desired state of the system in a declarative manner and Kubernetes is responsible for maintaining the desired state.

[1] `https://dse.in.tum.de`

Kubernetes uses a control loop to monitor the state of the system and make changes to the system to maintain the desired state. Kubernetes employs a consensus algorithm to ensure system consistency and fault tolerance in the presence of failures in a large distributed system.

kube-scheduler is the default scheduler component of Kubernetes, responsible for scheduling pods onto nodes. It uses an extensible scheduling plugin pipeline to determine the best node for each pod.

The default scheduling algorithm uses a priority-based approach to select the best node for each pod [38]. Kata Containers [31] is an open-source container runtime that uses lightweight virtual machines (VMs) to isolate containers. kata-urunc is a container runtime that extends Kata Containers to isolate unikernels in VMs using Funky Monitor to provision and manage unikernels. Funky also uses OpenFaaS, an open-source serverless framework for Kubernetes.

3 Overview

3.1 System Overview

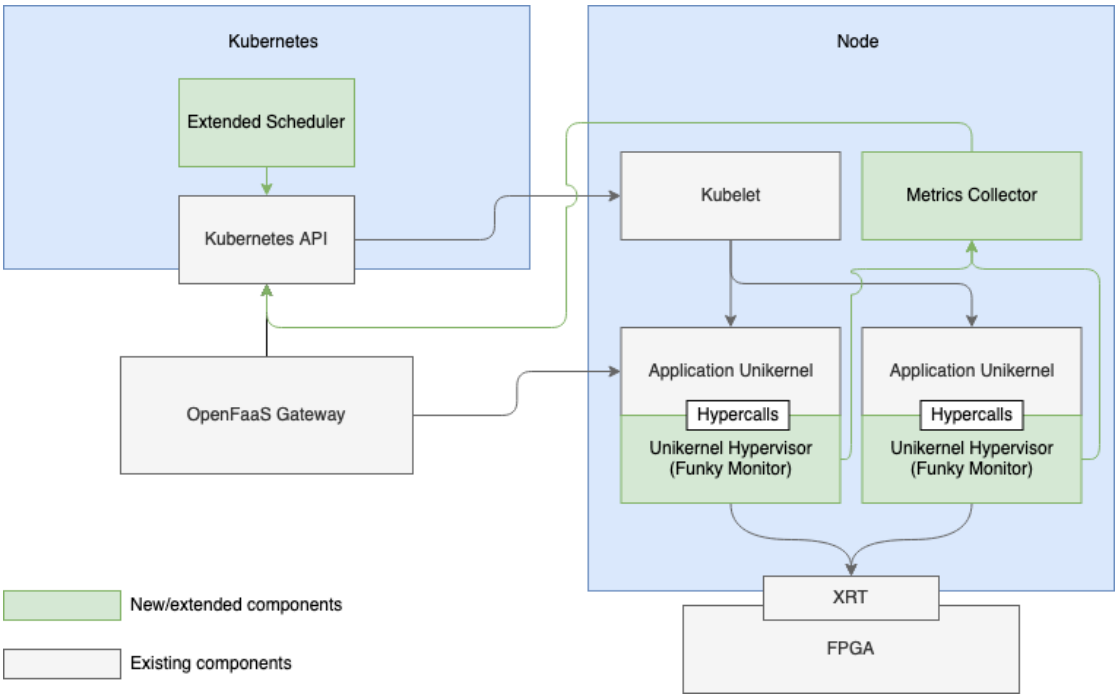


Figure 3.1: System Overview

We propose extending an existing serverless platform stack with FPGA awareness. Our system (shown in Figure 3.1) consists of compute nodes equipped with Alveo U50 FPGAs connected over PCIe and addressable by the host using the XRT API. Containers are deployed onto the nodes and managed by Kubernetes. The Kubernetes scheduler, kube-scheduler, is responsible for making placement decisions for incoming pod deployments. Internally, the Kubernetes scheduler relies on multiple plugins at specific scheduling stages (so-called extension points), including a new *fpgascheduling* plugin that places new pods based on metrics reported by the nodes. OpenFaaS is used to deploy and run functions on the cluster, as well as accepting function

invocations which are then routed to the respective function deployment. On the node, a metrics collector is responsible for processing and relaying FPGA-related metrics to the scheduler for improved placement decisions based on real usage data. Each function unikernel is deployed in a dedicated VM, which is managed by Funky Monitor. Funky Monitor adds hypercalls for acquiring, executing code on, and releasing FPGA slots on behalf of the application. Hypercalls are function calls from the unikernel to the hypervisor. They are used to perform operations that are not supported by the unikernel, such as memory allocation and I/O operations. Funky Monitor is extended to measure and report FPGA usage and reconfiguration metrics to the metrics collector.

3.2 Design Goals

This project focuses on extending an existing FaaS platform to support FPGA acceleration. The goal is to enable the efficient use of FPGA resources in a multi-tenant environment by making informed scheduling decisions based on real usage data. To this end, we aim to:

- Increase the utilization of FPGAs for serverless workloads, reduce FPGA reconfigurations, and ensure fairness between functions
- Create an extensible scheduling framework for FPGAs in serverless computing
- Keep a low scheduling overhead for fast scheduling decisions

The main challenges we face are extending the existing system stack, ensuring compatibility with existing deployments, and minimizing the impact on the overall system performance. Furthermore, we need to ensure that the proposed system can be used in a large-scale deployment.

3.3 System Workflow

The production system is split into two main workflows: deployment and invocation. The deployment workflow is responsible for deploying new functions to the cluster, while the invocation workflow handles incoming function invocations. Both workflows are extended to enable FPGA-accelerated functions. While deployment can also happen at invocation time, we will generally assume that functions are deployed before they are invoked.

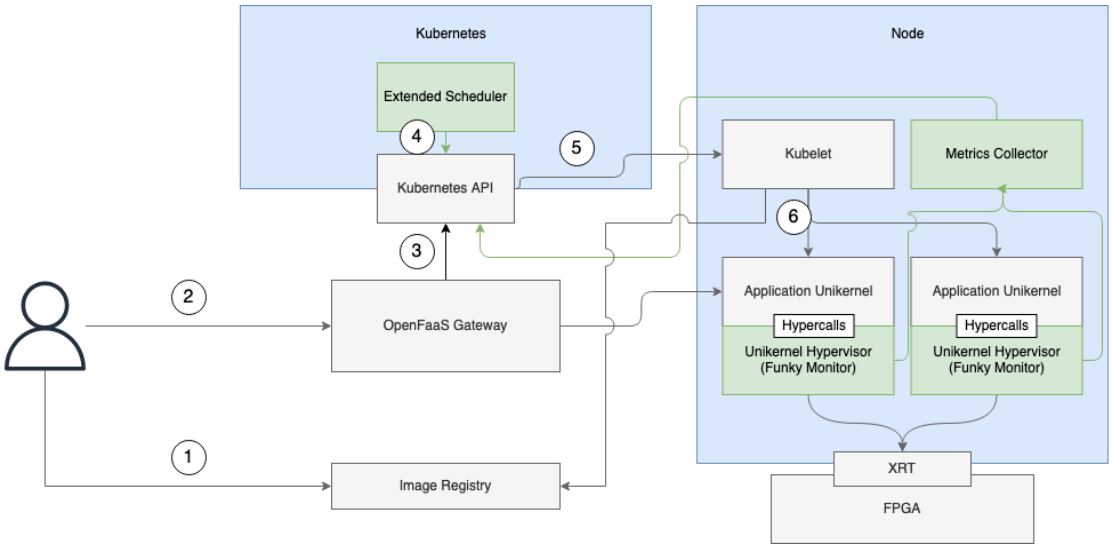


Figure 3.2: Serverless function deployment

3.3.1 Function Deployment

Before deploying a function, the user prepares a function image including compiled bitstreams for different FPGA configurations (See Figure 3.2). The image is uploaded to a registry (1), after which the user deploys the function to the serverless platform using the OpenFaaS API (2). During the deployment, the OpenFaaS controller creates the underlying Kubernetes deployment (3), which leads to pods being placed in the scheduling queue. The Kubernetes scheduler picks up the pods and computes the most suitable node to place each function on (4). In this step, the *fpgascheduling* plugin is used to calculate a score based on recent FPGA usage metrics. Once placed, the function pod is provisioned on the node by kata-urunc, which is invoked by the Kubernetes agent (kubelet) via the Container Runtime Interface (CRI) (5). The function unikernel is then started and hypercalls are intercepted by Funky Monitor (6).

3.3.2 Function Invocation

Every function invocation is received by the OpenFaaS gateway (See Figure 3.3) (1). The gateway forwards the invocation to the respective function pod, optionally performing load balancing in case multiple replicas exist (2). In case the function requires hardware acceleration, it performs a hypercall (3). The hypercall is intercepted by Funky Monitor, which configures the FPGA with the provided bitstream, if not already configured (4).

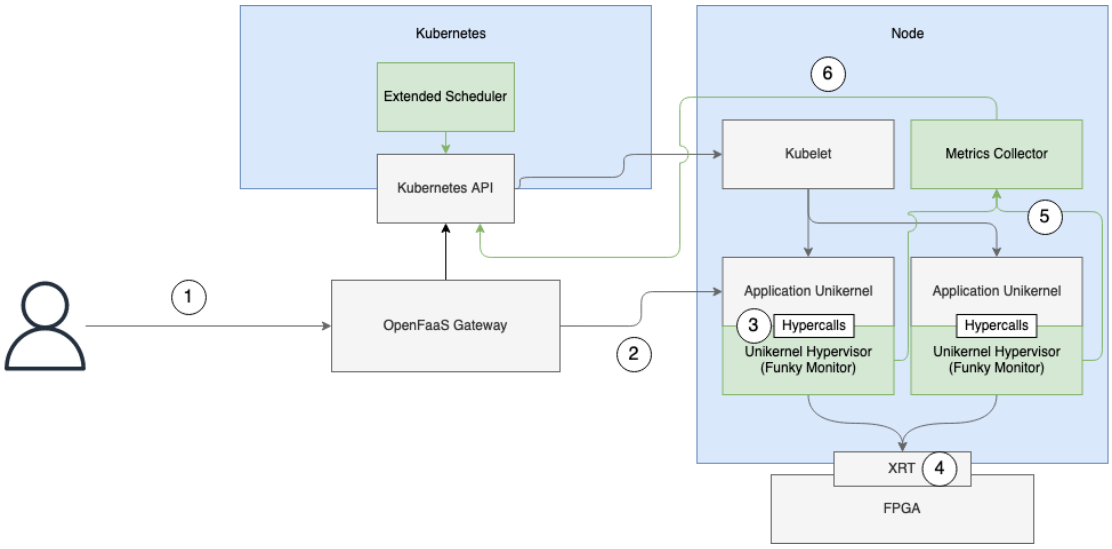


Figure 3.3: Serverless function invocation

The FPGA logic is then executed and usage is measured in the extended hypervisor, after which it is reported to the metrics collector (5). The metrics collector processes the received events and updates internal data structures tracking the system utilization, which are then shared with the Kubernetes cluster (6).

3.4 System Components

For each pod to be deployed, the **Kubernetes scheduler** is invoked to select the most feasible node to place the pod on. The scheduler first determines feasible nodes and then scores each node based on multiple criteria, including the scores determined by our proposed *fpgascheduling* plugin. **OpenFaaS** is used to abstract away the deployment and execution of functions for developers. The OpenFaaS controller is responsible for creating Kubernetes deployments for each function, which are then picked up by the scheduler. The OpenFaaS gateway is responsible for routing incoming function invocations to the respective function deployment. Kata Containers and the kata-urunc extension provide a container runtime compatible with the CRI required by Kubernetes to run function pods as isolated unikernels. **Funky Monitor** is extended to report metrics to the metrics collector. The **metrics collector** receives usage data from the hypervisor and reports them to the scheduler, by updating its internal state and annotating aggregate metrics to the Kubernetes node objects, which are accessible to the scheduler.

4 Design

4.1 System Design

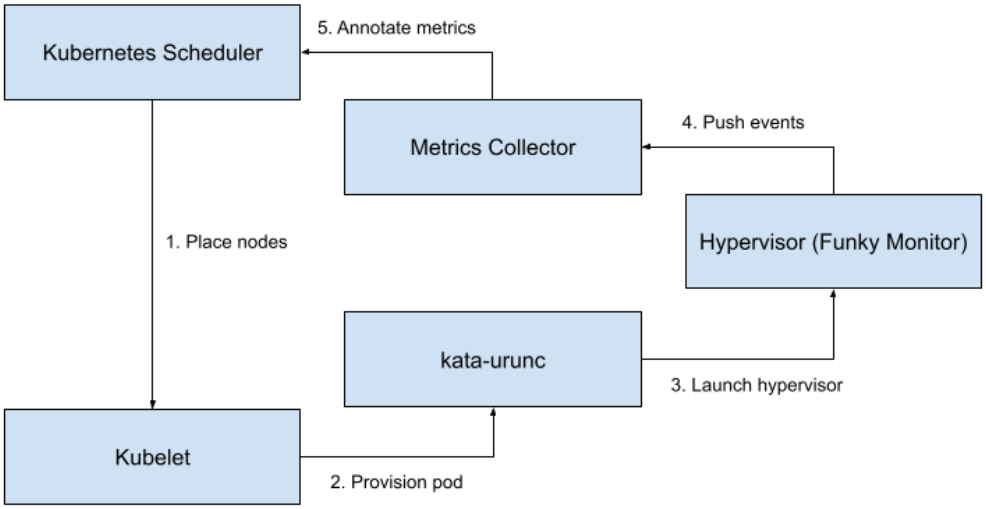


Figure 4.1: System communication flows

For a robust and extensible foundation, we use Kubernetes as the orchestration layer for our system. By default, the Kubernetes scheduler is not aware of FPGAs and cannot take them into account when scheduling pods. We extend the scheduler with a new *fpgascheduling* plugin to be aware of FPGAs and to schedule pods to nodes considering the FPGA utilization, recent reconfigurations, bitstream locality, and more. To improve placement decisions, we collect metrics about FPGA usage and resource availability using a metrics collector service deployed on each node. The metrics collector maintains an up-to-date model of the FPGA resource available on the node. We decouple the metrics collector from the scheduler using Kubernetes annotations as a shared key-value data store (See Figure 4.1). This allows updating the scheduling logic independently from the metrics collection. We use the aggregated

metrics to predict the frequency and duration of FPGA reconfigurations, cold starts, and resource utilization trends. For providing a fully-featured serverless experience, we use OpenFaaS as the underlying serverless platform. Furthermore, we use an extended Kata Containers runtime to run functions in a lightweight unikernel VM. Funky Monitor is the lightweight hypervisor based on ukvm/solo5 used to run the unikernel, intercepting hypercalls by the application for managing FPGA slots. We extend Funky Monitor to report FPGA metrics to the metrics collector. While we design the system to use Funky Monitor, we decouple the metrics collector from the hypervisor layer to enable reporting for other possible runtimes.

While many serverless platforms scale down idle functions after a sustained period without invocations, we do not assume scale to zero for the scope of this project. Additionally, instead of allocating FPGA slots at deploy time, we allocate them at invocation time when they are required, making the most efficient use of a fixed number of FPGA slots per node. When bitstreams are not yet configured for a function, acquiring FPGA capacities at runtime leads to cold starts, which we aim to reduce through better placement decisions made by the scheduler. We also do not assume preemption in our scheduling algorithm, as saving and restoring the system state in user-defined FPGA bitstreams requires additional logic and a compatible shell [34]. The majority of serverless function invocations and thus FPGA requests have been shown to execute in the order of a few seconds [60], at which point preemption might have negative effects on the overall performance. We expect most functions to be invoked infrequently with a small minority of functions receiving more than one invocation per minute [60]. Building on these real-world observations, we design our system to optimize function placement to reflect the invocation patterns to support the allocation of scarce FPGA resources at the highest possible resource utilization.

4.2 Kubernetes Scheduling

The scheduler assigns pods to the most feasible node by running through a scheduling pipeline as shown in Figure 4.2. New pods that are not yet assigned to a node are added to the scheduling queue [38] and picked up by the scheduler. Pods in the scheduling queue are sorted by priority and then by creation time. For each pod, the scheduler performs two steps: filtering and scoring. In the filtering step, given a list of nodes, the scheduler filters out infeasible nodes, i.e., nodes that cannot run the pod. In the scoring step, the scheduler ranks the remaining nodes based on a scoring function. The scheduler then assigns the pod to the highest-scoring node. For each of the scheduling lifecycle steps, the scheduler exposes extension points that can be used to extend the default behavior. In the standard configuration, the kube-scheduler configuration uses

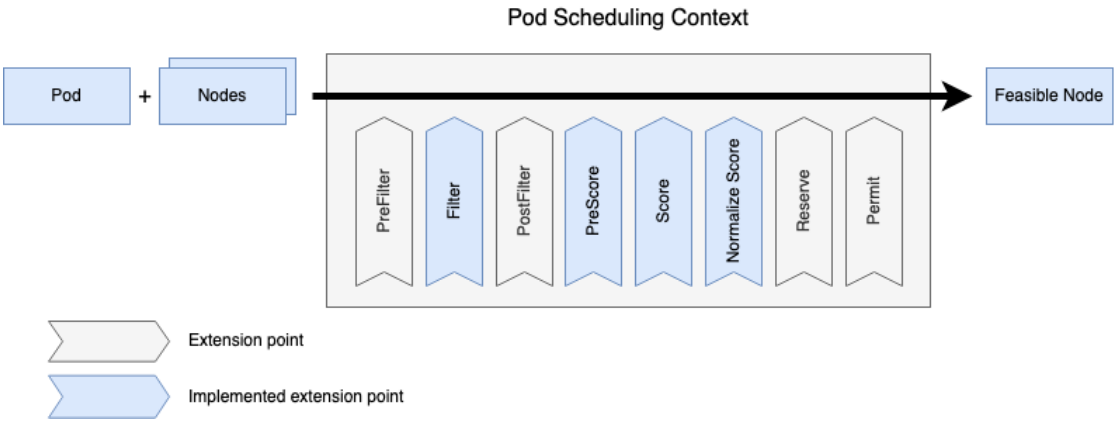


Figure 4.2: Extension points exposed by scheduling framework [58].

pre-defined plugins, including ImageLocality (scoring nodes based on container image availability), NodeResourcesFit (filtering nodes based on CPU and memory availability), and NodeAffinity (scoring nodes based on node labels).

We propose a new plugin extending the filtering and scoring steps to take FPGA metrics into account to improve resource utilization and fairness. First, nodes that do not meet the criteria required by the workload such as FPGA vendor are filtered in the *Filter* extension point. In the *PreScore* lifecycle step, the scheduling plugin receives a list of filtered nodes, and the current pod to be scheduled. The scheduling plugin computes a score (See Figure 4.3) for each node based on the reported metrics.

We design the score as a weighted average of multiple criteria, including the recent usage in seconds (lower is better), recent reconfiguration time in seconds (lower is better), and bitstream locality (1 if FPGA has recently configured bitstream required by function, otherwise 0). First, we compute the score by calculating the relative position compared to the best value measured for each metric (the best value receives a score of 1, following values are relative to the best value). We then weigh the scores for each metric by a configurable weight and compute the weighted average, normalizing by the sum of the weights to ensure that the score is in the range of 0 to 1. The weights can be configured upfront (See Table 4.1) by the cluster administrator to reflect the desired trade-off between the different metrics. In the following, scheduler weights will be shortened to the format `<bitstream locality weight>-<usage time weight>-<reconfiguration time weight>`.

To avoid putting heavy computation in the critical path, the plugin will make decisions based on metrics computed by the metrics collector in the background, achieving minimal scheduling overhead. The metrics are then stored in Kubernetes

```
def computeNodeScore(
    sortedUsageTimes: List[float], recentUsageTime: float,
    sortedReconfigTimes: List[float], recentReconfigurationTime: float,
    hasFittingBitstream: bool,
    RecentUsageTimeWeight: float, RecentReconfigurationTimeWeight: float,
    BitstreamLocalityWeight: float
) {
    usagePosition = sortedUsageTimes.index(recentUsageTime)
    reconfigurationPosition = sortedReconfigTimes.index(recentReconfigurationTime)

    usageScore = ((len(sortedUsageTimes) - usagePosition) / len(sortedUsageTimes))
    reconfigurationScore = (
        (len(sortedReconfigTimes) - reconfigurationPosition) /
        len(sortedReconfigTimes)
    )

    return (
        (usageScore * RecentUsageTimeWeight) +
        (reconfigurationScore * RecentReconfigurationTimeWeight) +
        (hasFittingBitstream * BitstreamLocalityWeight)
    ) / (
        RecentUsageTimeWeight + RecentReconfigurationTimeWeight +
        BitstreamLocalityWeight
    )
}
```

Figure 4.3: Computing node scores using weighted average

Name	**Description**
`RecentUsageTimeWeight`	Recent FPGA usage time
`RecentReconfigurationTimeWeight`	Recent FPGA reconfiguration time
`BitstreamLocalityWeight`	Place on node with configured bitstream

Table 4.1: Scheduler plugin configuration options, weights are floating point numbers

annotations, accessible by the scheduler plugin. Annotations are key-value pairs that can be attached to Kubernetes objects and may contain arbitrary data. The annotations are used as a decoupling point between the metrics collector and the scheduler plugin and provide a common data interface between the two components. This allows us to

easily push metrics from a different source and use the provided metrics for scheduling decisions by altering the scheduler plugin only.

4.3 Function Lifecycle

The primary unit of interest in our system is the function, a packaged user workload designed to run on the CPU and accelerated by an FPGA when required. During deployment, the scheduler is in control to determine the best possible placement for a given function. This step is crucial for ensuring uniform resource utilization throughout the cluster. At invocation time, a function may acquire an FPGA slot to perform accelerated computations, which may require reconfiguration of the FPGA. The hypervisor is responsible for ensuring fairness and preventing starvation among competing workloads on the same node. This project focuses on the scheduler as a key enabler for FPGA acceleration in serverless environments. We extend the hypervisor to support FPGA acceleration and fairness but do not cover the remaining implementation details of the hypervisor.

4.3.1 Function Deployment

Ahead of time, the developer deploys the function to the cluster using the OpenFaaS API. Creating a deployment in turn creates a pod for the function. The pod is scheduled to a node by the Kubernetes scheduler using our extended scheduler. At that point, our *fpgascheduling* plugin is invoked (See Section 4.2). The plugin retrieves the latest utilization and usage metrics from the node annotations, which does not require a further network roundtrip. After returning the score, the scheduling framework computes the weighted average score for all plugins and binds the pod to the highest-scoring node. Once assigned to a node, the Kubernetes agent running on that node attempts to provision the container by invoking the applicable container runtime, in our case the extended Kata Containers runtime, using the CRI. kata-urunc in turn launches a Funky Monitor instance configured to boot the user image and bitstream contained in the Open Container Initiative (OCI) Image. Once launched, the function waits for incoming invocations or events.

4.3.2 Function Invocation

When FPGA acceleration is required for processing a received request, the application unikernel performs a *fpgainit* hypercall. The hypervisor intercepts the hypercall and forwards the request to the FPGA device using OpenCL and XRT. The final duration of the FPGA requests and the reconfiguration time are periodically reported to the

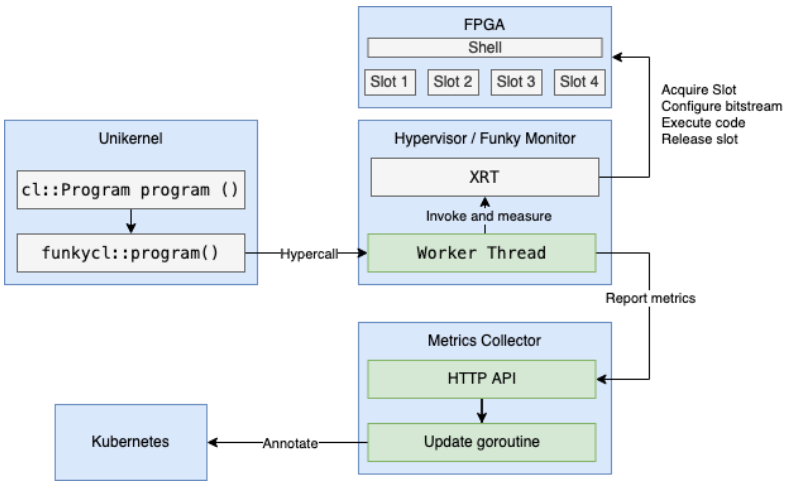


Figure 4.4: Detailed flow from OpenCL invocation to FPGA reconfiguration by the hypervisor and metrics collection, reporting, and annotation by the metrics collector.

metrics collector to inform the scheduler about the current system utilization. This way, future deployments are based on recent system utilization and can be scheduled onto nodes with sufficient FPGA resources, balancing the load and FPGA utilization across the cluster. A detailed flow can be seen in Figure 4.4, showing the initial invocation of OpenCL, which is extended by the Funky project to send a hypercall to Funky Monitor. In the hypervisor, OpenCL is used with the XRT backend to configure and execute code on the FPGA.

4.4 System Components

4.4.1 Kubernetes Scheduler Plugin

Not only is the default Kubernetes scheduler [36] limited to CPU and memory resources when scheduling pods, the allocated resources are not matched against actual utilization but determined relative to the total available resources on a node. This approach is insufficient for scheduling FPGA-enabled serverless workloads. While extending Kubernetes to add FPGA slots as an extended resource similar to CPU and memory would allow a static allocation of FPGA slots, this design would inevitably lead to potential underutilization of available FPGAs and un-schedulable workloads in case the desired number of functions were higher than the number of available slots. Because of this, we propose a scheduler plugin adding FPGA awareness to Kubernetes in a

minimally-invasive way. For each node, we maintain a model of the FPGA resource utilization and availability, which is kept in sync by the metrics collector service. When a new pod is added to the scheduling queue, we consider the FPGA utilization for the placement decision as shown in Figure 4.3, as well as matching the FPGA vendor and shell version required by the function to a node equipped with a fitting accelerator.

The scheduler can be extended with additional criteria in the future, such as package-aware scheduling, bitstream size considerations, or FPGA model preferences. Scheduler weights can be configured upfront and can be adjusted for each available metric (See Table 4.1), including recent FPGA usage time, recent FPGA reconfiguration time, and bitstream locality.

4.4.2 Metrics Collector

The metrics collector is a Go service deployed per node and receives requests from all hypervisor instances running on that node. It aggregates the received metrics and periodically reports them to the scheduler using a shared data structure (See Figure 4.5) by annotating the Kubernetes node object. The metrics collector exposes an HTTP endpoint for receiving application metrics including FPGA usage and reconfiguration events pushed from the hypervisor.

```
type FPGANodeState struct {
    RecentUsage float64
    RecentReconfigurationTime float64
    RecentBitstreamIdentifiers map[string]struct{}
}
```

Figure 4.5: FPGA Node State

4.4.3 Hypervisor Metrics Reporter

Per unikernel application, one instance of Funky Monitor is started on the host machine. Funky Monitor forwards the FPGA requests to the real FPGA device using OpenCL and the XRT. We intercept all FPGA requests and report them to the metrics collector, measuring the duration and reconfigurations on the hypervisor level.

A background worker thread is launched for communicating with the FPGA, reconfiguring the FPGA if necessary, sending data to the FPGA, and executing programmed kernels through OpenCL.

5 Implementation

The implementation of the system is split into three main components: the unikernel hypervisor metrics reporter, the metrics collector, and the Kubernetes scheduler extension. Additional implementation work has gone into the simulator, which is covered in Chapter 6. The source code for the scheduler extension, the metrics collector, as well as the simulator is available on GitHub [1].

5.1 Hypervisor Metrics Reporter

```
auto fpga_worker_thread = [](...)
{
  // create span and start measuring fpga workload duration
  cPerfTimer *timer = new cPerfTimer();
  timer->start();

  // initialize worker and reconfigure FPGA if necessary
  funky_backend::Worker worker(thr_info, rq_addr, wq_addr);
  double reconfigured = worker.reconfigure_fpga();
  if (reconfigured > 0) {
    add_dispatcher_request({
      .collector = thr_info.metrics_collector,
      .timestamp_ms = get_time_ms(),
      .duration_ms = 0, // usage is reported in a dedicated request
      .reconfiguration_ms = reconfigured,
      .bitstream_identifier = thr_info.bitstream_identifier
    });
  }
```

Figure 5.1: Starting timers and recording reconfiguration events. Usage time is reported in a dedicated request subsequently

[1] https://github.com/BrunoScheufler/serverless-fpga

To enable a multi-tenant environment with multiple applications deployed on each node, the FPGA is partitioned into multiple virtual slots. Serverless functions are deployed ahead of time before they are invoked. This allows reacting to incoming requests faster but potentially requires more idle resources. Each application is deployed as a unikernel with its own Funky Monitor instance. During an invocation, the application decides whether it needs FPGA acceleration or not in the host process. When the application sends an `fpgainit` hypercall to acquire a slot, the hypervisor initializes a worker thread, if not already running (See Figure 5.1), and forwards the request to the FPGA. `worker.reconfigure_fpga()` is a blocking call that invokes OpenCL with the XRT backend to place the bitstream and reconfigure the FPGA if necessary. This dynamic allocation of slots is similar to a connection pool, where each application can request a slot when it needs to execute a function. When the function is completed, the slot is released and can be used by another application.

We extend the creation of the worker thread with timing calls to measure the duration each invocation spends computing on the FPGA, as well as tracking reconfigurations. Internally, we detect reconfigurations by measuring the delay of invoking XRT API functions, as the XRT does not expose reconfiguration events. We measure the duration as the complete time from thread initialization to thread termination, which includes FPGA reconfiguration and forwarding requests to the FPGA. To measure timings, we use a high-resolution clock provided by the Linux kernel, which is available in the `<chrono>` header of the C++ standard library.

```
timer->stop();
add_dispatcher_request({
  .collector = thr_info.metrics_collector,
  .timestamp_ms = get_time_ms(),
  .duration_ms = timer->get_ms(),
  .reconfiguration_ms = 0,
  .bitstream_identifier = thr_info.bitstream_identifier
});
```

Figure 5.2: Collecting request timing

Metrics are lazily collected throughout the session, out of the critical path to avoid degrading the performance of the application. For this, we use a separate background thread to collect metrics in a standard double-ended queue provided by the C++ standard library. The metrics dispatcher thread is started during the hypervisor startup and runs until the hypervisor is shut down. This way, recording metrics and traces is as simple as invoking `add_dispatcher_request()` and passing the respective parameters.

5.2 Metrics Collector

The metrics collector is a Go service running on each node, accepting requests from the Funky Monitor instances. It aggregates metrics from all Funky Monitor instances and updates internal data structures for each node. Metrics updates are synchronized with the Kubernetes scheduler by updating node annotations using the Kubernetes API. Internally, the metrics collector records FPGA usage and reconfigurations reported by the hypervisor instances using priority queues and a sliding window to keep track of the most recent metrics. Periodically, requests older than the sliding window are removed from the queue as shown in Figure 5.3, and the node annotations are updated. The priority queues are serialized using binary encoding (gob) and included in the annotation to allow the collector to recover from crashes and restarts.

```
SLIDING_WINDOW := 5 * time.Minute

func (tracker *UsageBasedFPGAUtilizationTracker) Decay() {
  for e := tracker.Requests.Front(); e != nil; {
    next := e.Next()
    req := e.Value.(*UsageRequest)
    if time.Now().Sub(req.Timestamp) > SLIDING_WINDOW {
      tracker.Requests.Remove(e)
      tracker.recentUsage -= time.Duration(req.Usage)
    }
    e = next
  }
}
```

Figure 5.3: Decaying older requests

5.3 Kubernetes Scheduler Plugin

The Kubernetes scheduler can be extended in multiple ways, using extender webhooks or by implementing custom plugins using the scheduling framework. We use the scheduling framework [58] to implement a plugin that enables FPGA-aware scheduling. Custom plugins are implemented as Go modules and can be compiled into the scheduler binary. Deploying a custom scheduler requires building an OCI Image based on the custom scheduler binary and deploying it to the Kubernetes cluster. Configurable scheduler weights can be supplied in the plugin configuration, as seen in Table 4.1.

The scheduler plugin attaches at the *Filter*, *PreScore*, *Score*, and *NormalizeScore* extension points exposed by the scheduling framework. *Filter* is invoked for every node and determines whether a given pod specification can be placed on the current node, identifying feasible nodes to be scored subsequently. Our *Filter* implementation checks whether the pod specification contains a `fpga-scheduling.io/fpga-vendor` label and, if so, filters out all nodes with different FPGA vendors (See Figure 5.4). While we only rule out nodes with different vendors, the *Filter* step can be expanded to additional criteria as laid out in Chapter 9.

```
func assertVendorLabel(
  ctx context.Context,
  state *framework.CycleState,
  pod *v1.Pod,
  nodeInfo *framework.NodeInfo,
) *framework.Status {
  vendorLabel, hasVendorLabel :=
    pod.Labels["fpga-scheduling.io/fpga-vendor"]
  if !hasVendorLabel {
    return nil
  }

  nodeVendorLabel, nodeHasVendorLabel :=
    nodeInfo.Node().Labels["fpga-scheduling.io/fpga-vendor"]
  if !nodeHasVendorLabel {
    return nil
  }

  if vendorLabel != nodeVendorLabel {
    return framework.NewStatus(
      framework.UnschedulableAndUnresolvable,
      "Pod␣requires␣a␣different␣FPGA␣vendor␣than␣the␣node",
    )
  }

  return nil
}
```

Figure 5.4: Filtering mismatching FPGA vendors

PreScore receives a list of nodes and the current pod to be scheduled as shown in Figure 5.5. While *PreScore* is not responsible for returning the scores, *Score* is invoked per node, while our scoring algorithm (See Figure 4.3) relies on having access to all node scores to calculate relative positions. Because of this, we opt to calculate all node scores in *PreScore* and store the result in the cycle state, which is passed to *Score*, where the individual node score is retrieved and returned. *NormalizeScore* has no effect as the scores are automatically normalized in the *PreScore* extension point.

```
func (pl *FPGAScheduling) PreScore(
  pCtx context.Context,
  cycleState *framework.CycleState,
  pod *v1.Pod, nodes []*v1.Node,
) *framework.Status {
  preScores := make([]nodePreScore, 0, len(nodes))
  for _, node := range nodes {
    preScores = append(preScores, calculatePreScoreForNode(node))
  }

  state := preScoreState{make(map[string]int64)}
  for _, preScore := range preScores {
    relativeReconfigurationScore,
      relativeUsageScore, hasFittingBitstream :=
        calculateRelativeReconfigurationScore(preScore)

      state.fpgaScore[preScore.nodeName] = getFinalScore(
        relativeReconfigurationScore,
        relativeUsageScore,
        hasFittingBitstream
      )
  }

  cycleState.Write(preScoreStateKey, &state)
  return nil
}
```

Figure 5.5: Calculating node scores in the PreScore extension point

6 Evaluation

To evaluate the contributions of this project, we consider the impact of the scheduler on system performance, more specifically on FPGA utilization and fairness. The Kubernetes scheduler is limited to placement decisions and performs a static allocation of pods to nodes. Because of this, the scheduler only has limited control of fairness related to FPGA access as this is controlled by the runtime on the node. The runtime, which is beyond the scope of this project, queues up requests from applications accessing the FPGA at the same time based on criteria such as priority and fairness. Even so, the scheduler is crucial for the distribution of workloads across nodes: A balanced allocation of workloads across nodes improves resource efficiency, increases system throughput, and reduces hotspots, while an unbalanced allocation can lead to underutilization of resources and performance degradation. A uniform workload distribution with roughly equal resource utilization leads to higher energy efficiency, less wasted compute resources, and lower costs [17][22][46][10].

To evaluate the extended Kubernetes scheduler, we consider the following high-level research questions:

- How does the scheduler perform compared to the default scheduler (baseline) in a simulated environment?
- Is the scheduler able to improve fairness and resource utilization of the system?
- How does the scheduler perform under different workloads?
- Which invocation patterns make the scheduler perform better/worse?

6.1 Experimental Testbed

We compare scheduler performance to the default scheduler implementation (the baseline) in a simulated environment, using production traces from a real-world serverless platform. To implement an evaluation framework, we differentiate between two possible evaluation models: Testing the scheduler in isolation and streaming requests into the system to evaluate the impact of metrics on scheduling decisions. Testing in isolation neither evaluates metrics collection nor is it designed for usage with production traces. However, it decouples deployment and invocation as in the

real system. Another significant challenge is defining the initial system state including resource utilization. Streaming request simulation evaluates the influence of metrics on the scheduler by streaming traces into the system. Using a continuous stream of requests, the scheduler makes placement decisions based on recorded metrics and updates metrics after each request, similar to the production system. A difference to the real implementation, however, is the just-in-time placement of applications as a reaction to function invocations with missing deployments. In this case, there is no decoupling between the deployment and invocation workflow, as in the real world. Streaming invocations builds up utilization metrics over time, which enables the simulation of the entire system using realistic workloads. In this project, we settle for the streaming request simulation as it covers the influence of the scheduler on the entire system performance and allows us to evaluate the scheduler in a large-scale deployment without requiring extensive hardware and system stack configuration.

We perform a Discrete-event simulation (DES) with next-event time progression by streaming pre-recorded traces into the simulator and handling requests serially. DES is a common approach to system simulation, in which the simulation advances in steps, where each step represents a discrete event, such as the start of a function invocation, the end of a function, a cold start, or a reconfiguration of an FPGA. This method allows capturing temporal aspects such as start times and durations from request traces.

While streaming requests into the simulated system, we continuously update the state of the system, including function assignment to nodes, FPGA slot allocation, and FPGA utilization and reconfiguration metrics used by the scheduler. We also record metrics used throughout the evaluation, such as the workload placement distribution and the number of reconfigurations per node.

6.2 Data Set

To evaluate the system on real-world data, we utilize production traces from Microsoft Azure Functions [72]. The data set contains traces of function invocations for two weeks starting on 2021-01-31, capturing invocation arrival and departure (or completion) times, as well as the respective functions. In total, 1.9 million requests, 119 distinct applications, and 424 distinct functions are recorded with a median of 415 requests per application, 2 functions per application, and 29 requests per function.

In Figure 6.1 we illustrate that half of the recorded functions in the traces data set (with at least one request) have fewer than one request per minute, while 75% of functions have fewer than 2 requests per minute. We have also visualized the arrival times in a timeline in Figure 10.1, which shows all invocations from a sample of 20 functions from [72]. We draw two key observations from the number of invocations and timeline:

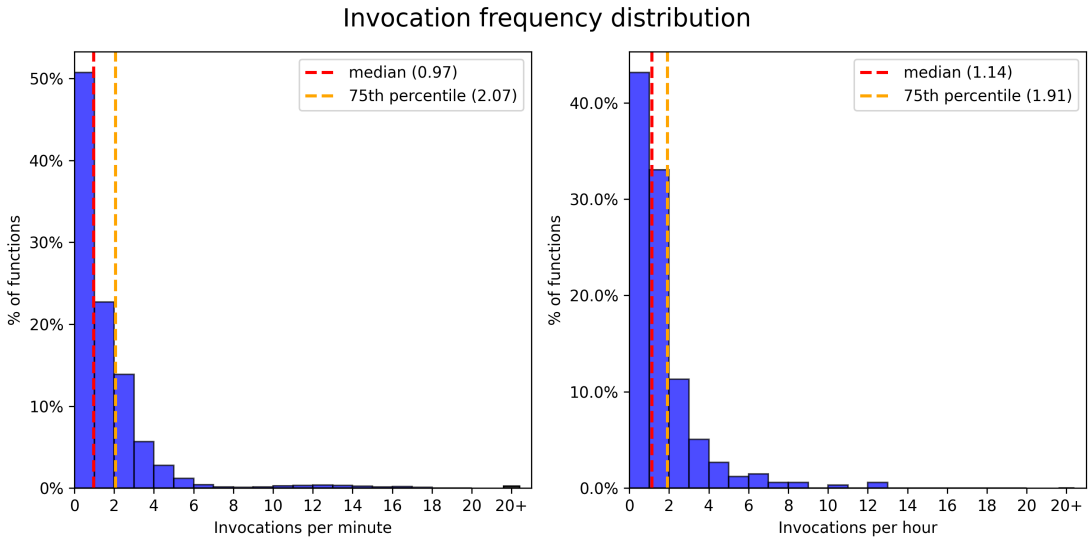


Figure 6.1: Histogram of number of invocations per minute/hour for [72]. All values exceeding the maximum value are grouped as aggregate into the last bin.

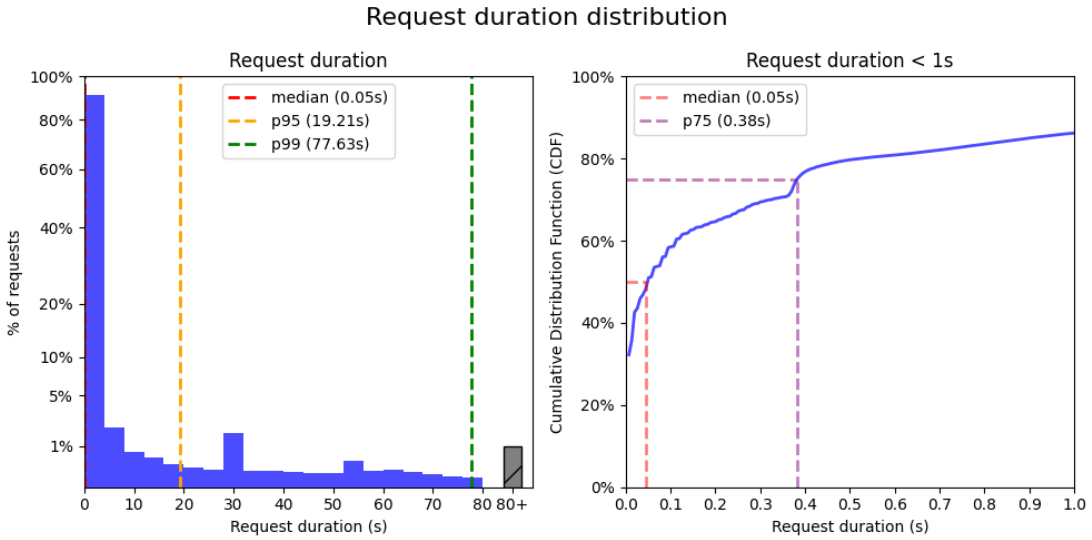


Figure 6.2: Histogram of request duration distribution [72]. For the first subplot, all values exceeding the maximum value are grouped as aggregate into the last bin.

The majority of functions are invoked infrequently, which is a common characteristic of serverless workloads [72], and some functions are invoked in bursts, with a large number of invocations happening in a short time frame, while others are invoked more evenly distributed over time. The low invocation frequency shows the significance of dynamically reusing FPGA slots configured for unpopular functions, which are expected to be idle most of the time, while the burstiness shows the importance of keeping bitstreams in place after an invocation if further invocations are expected shortly.

Furthermore, in Figure 6.2, we have illustrated the distribution of invocation durations showing a median duration of 46ms, with 95% of invocations being shorter than 19.21s and 99% of invocations being shorter than 77.63s. The second subplot of Figure 6.2 depicts the cumulative distribution function (CDF) of the request duration distribution for invocations shorter than one second. For example, 75% of invocations are shorter than 382ms. This shows that the majority of requests contained in the recorded traces from [72] are short-lived, guiding our design decisions toward reducing the overhead of reconfiguring FPGAs.

6.3 Methodology

To evaluate the system in a reasonable real-world scenario, we first explore the required system parameters, including system constants such as FPGA reconfiguration time, the number of nodes and FPGA slots per node, scheduling weights, and ratios of time spent on the host CPU and FPGA as well as requests in need of hardware acceleration considering both the percentage of all invocations and minimum invocation duration. We keep the underlying data set as is, to prevent biasing the results towards a specific synthetic workload and to allow for comparison with existing workloads. Once the system parameters are defined, we move on to evaluating the scheduling performance in the simulated system.

6.4 FPGA-related Constants

While varying designs are feasible for FPGA-accelerated serverless workloads, we define a constant number of four FPGA slots per device as well as a reconfiguration time of 10ms for configuring a bitstream using partial reconfiguration. These values have been aligned with recent work [35][55] and the proposed Funky project.

6.5 Characterizing Heterogeneous Workloads

Before we evaluate the scheduling performance, it is important to understand the impact of using different utilization measurements for scheduling serverless workloads. With the metrics collector, we are tracking FPGA utilization in addition to existing metrics for CPU, memory, and other compute resources. Our proposed *fpgascheduling* plugin uses the provided metrics to make informed placement decisions.

We expect our scheduling plugin to improve the distribution of workloads across nodes, leading to higher utilization of FPGA hardware. This effect is most apparent when system FPGA utilization diverges from CPU utilization, making the extended scheduler choose a different node than it otherwise would, considering only CPU and memory utilization. Because of that, we need to understand the underlying nature of heterogeneous computing workloads.

Each incoming function invocation is received by the respective application deployment and processed on the host CPU. If hardware acceleration is deemed beneficial to completing the request, the application sends a hypercall to Funky Monitor, offloading computationally intensive parts of the application to an available FPGA slot. In the time a function spends waiting on an FPGA, it can receive and process further requests or idle until the FPGA has completed the computation. However, because of this system design, every invocation requires CPU time for receiving and processing the request, therefore CPU utilization will always increase with FPGA utilization, though not necessarily at the same rate (the more an application benefits from FPGA acceleration, the lower the required CPU time). Not all requests require hardware acceleration, and the ones that benefit from it spend varying amounts of time on the FPGA.

The underlying traces provided by Microsoft Research [72] used to evaluate the system are not designed for heterogeneous computing, so do not specify which processor types were used to handle the request and for how long they ran. To our best knowledge, there is no existing data set that captures the distribution of CPU and FPGA utilization for individual serverless applications, requiring us to create a reasonable approximation. For the following system simulation, we, therefore, explore different ratios of CPU and FPGA utilization per request, as well as fixed ratios of requests using FPGA acceleration and minimum invocation time required for considering requests eligible for acceleration, allowing us to enhance the existing data set for evaluation purposes.

To understand the most fitting serverless workloads for our scheduler, we compared the baseline utilization rate of change with the FPGA utilization rate of change for varying FPGA usage ratios as defined above. A stronger divergence of recorded utilization indicates that FPGA utilization-aware scheduling can improve the overall system utilization, thus defining invocation patterns more suitable for our scheduler.

In Figure 6.5, we explore the development of FPGA and CPU utilization in multiple simulator runs, adjusting for different system parameters, including the percentage of invocations requiring hardware acceleration (p), the minimum invocation duration required for FPGA acceleration (m) and the ratio of invocation time spent on the FPGA vs. the CPU (r). Using these parameters, we can enhance the existing data set to cover real-world serverless workloads with FPGA acceleration to understand which processor types are predominantly used.

p r	0.00	0.25	0.50	0.75	1.00
0.00	0.00	0.00	0.00	0.00	0.00
0.25	0.00	0.14	1.23	3.80	8.09
0.50	0.00	0.84	3.09	8.27	17.16
0.75	0.00	0.36	3.07	9.09	24.23
1.00	0.00	10.37	21.79	30.48	28.23

Figure 6.3: m=0.01 (10ms)

p r	0.00	0.25	0.50	0.75	1.00
0.00	0.00	0.00	0.00	0.00	0.00
0.25	0.00	0.09	0.77	2.88	6.50
0.50	0.00	0.65	2.09	6.09	14.62
0.75	0.00	0.15	1.66	6.94	19.73
1.00	0.00	0.59	2.15	6.52	3.53

Figure 6.4: m=0.1 (100ms)

Figure 6.5: Difference between system FPGA and CPU utilization (in seconds) for different values of p and r with a minimum invocation duration of m seconds. Negative values indicate that the CPU utilization is higher than the FPGA utilization.

From the results, we can observe that the more requests are performed with FPGA acceleration (higher p) and the more time is spent per invocation (higher r), the more important it becomes to schedule workloads informed by recent FPGA utilization data. On the other hand, when most invocations are executed on the host CPU (p) and only a few invocations even require hardware acceleration (low r), FPGA utilization will lag behind the CPU utilization. Setting a lower threshold for FPGA acceleration (lower m), the higher the FPGA utilization if these requests spend a significant amount of time on the FPGA (higher r). However, if we set the threshold m too high, most requests will not be eligible for acceleration, leading to a lower FPGA utilization. This effect can be seen in the second table, where the CPU utilization always outweighs the FPGA utilization, as 60% of invocations are faster than 100ms and therefore not eligible for acceleration.

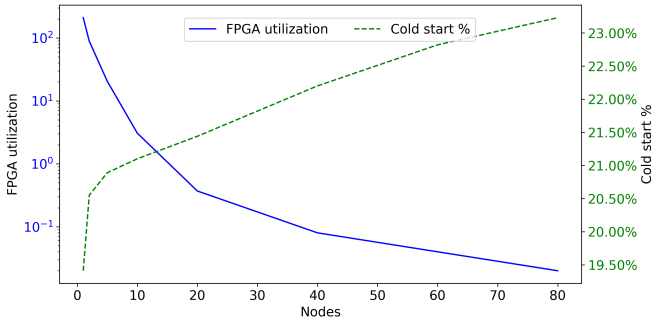
Given an average partial reconfiguration time between 10ms and 20ms [35], we can assume that a minimum invocation duration of **10ms** is a reasonable lower bound and value for m, as invocations shorter than 10ms would not benefit from FPGA acceleration in case of a cold start. At the same time, setting m to a higher value severely limits the eligible invocations. Finding a reasonable value for p highly depends on the expected workloads in the system: If the system is designed for mostly CPU-only functions

with infrequent accelerated invocations in a multi-tenant environment, only a few applications will support acceleration. However, in the case of a dedicated system for accelerated applications, most invocations will make use of acceleration capabilities. To support mixed workloads we, therefore, assume a value of **0.5** for p as a reasonable middle ground. Lastly, r depends on the application and the amount of acceleration it can benefit from. The more business logic is moved to the FPGA, the higher the ratio will be. We assume a value of **0.75** for r, as we expect eligible invocations to spend most of their time on the FPGA.

Please keep in mind that the actual performance of the scheduler depends on the underlying workload and the system parameters. The above assumptions are based on the results of the system simulation and may not hold for all workloads.

6.6 Number of Nodes

While large-scale production systems rely on tens of thousands of distributed nodes for processing requests, serving the 424 distinct functions contained in the data set requires only a fraction thereof.



Nodes	FPGA utilization	Cold start %
1	210.01	19.41
2	88.88	20.55
5	20.52	20.89
10	3.07	21.10
20	0.37	21.44
40	0.08	22.20
60	0.04	22.82
80	0.02	23.23

Figure 6.6: Resulting median FPGA utilization in seconds and cold start percentage depending on the number of nodes for equal scheduler weights.

Deciding on the optimal number of nodes is a trade-off between a higher distribution of workloads and fewer cold starts but ultimately the risk of under-utilization of resources while too few nodes lead to potential noisy neighbor effects, more cold starts, and higher resource utilization to the point of over-utilization and performance degradation. Fundamentally, the argument is a financial one: The more nodes are used, and the lower the utilization, the higher the cost of the system. The best number of nodes is therefore the one that minimizes the cost of the system while still providing the required performance.

To settle on a reasonable number of nodes, we perform a system simulation run with varying numbers of nodes and find the acceptable trade-off. In Figure 6.6, we can see that increasing the number of nodes drastically reduces the median FPGA utilization (going from 20.52s with 5 nodes to 3.07s with 10 nodes). However, the cold start percentage slightly increases counter to our expectations, as the scheduling goals of equal distribution and bitstream locality clash, leading to functions and their bitstreams being distributed more evenly and thus requiring additional cold starts. Results of an additional run preferring bitstream locality weighted 8:2 over the remaining criteria are added in Figure 10.2 and confirm a steadily lower cold start percentage of 19.41% in case of higher adherence to bitstream locality.

For the remainder of this project, we will use 10 nodes as a baseline for our system design.

6.7 Scheduler Weights

As we have observed before, the choice of scheduler weights impacts the results considerably. Subsequent invocations of the same function are likely to use the same bitstream. Assuming we keep the bitstream loaded onto the device, placing functions with the same bitstream on the same node avoids reconfiguration overhead. The more frequent requests for distinct functions with different bitstreams occur, the higher the chance grows that a reconfiguration is required. If popular functions can remain on the same node with few competing functions in terms of request frequency, the reconfiguration overhead can be reduced. Thus, increasing the bitstream locality weight leads to comparatively lower reconfigurations at the cost of a higher concentration, whereas higher recent FPGA usage and reconfiguration time weights lead to a more distributed placement of applications while increasing the number of reconfigurations.

In Figure 6.7, we run multiple simulations with different scheduler weights. While equal scheduler weights offer a good starting point with a median FPGA utilization of around 3.07s and a cold start percentage of 21.10%, increasing the weights related to FPGA usage to 40% each while reducing bitstream locality to account for 20% of the total score (i.e. a scheduler weight of 1-2-2) yields improvements on both ends, increasing the median FPGA utilization to 4.90s while decreasing the cold start percentage to 20.69%.

Further increasing the weights to 1-3-3 and beyond increases the median FPGA utilization as expected but increases the cold start percentage as well, indicating that the scheduler is unable to find a good balance between the two goals. On the other end of the spectrum, increasing the bitstream locality weight to 66.67% while reducing the remaining weights to 16.67% each (scheduler weights of 4-1-1) yields a median

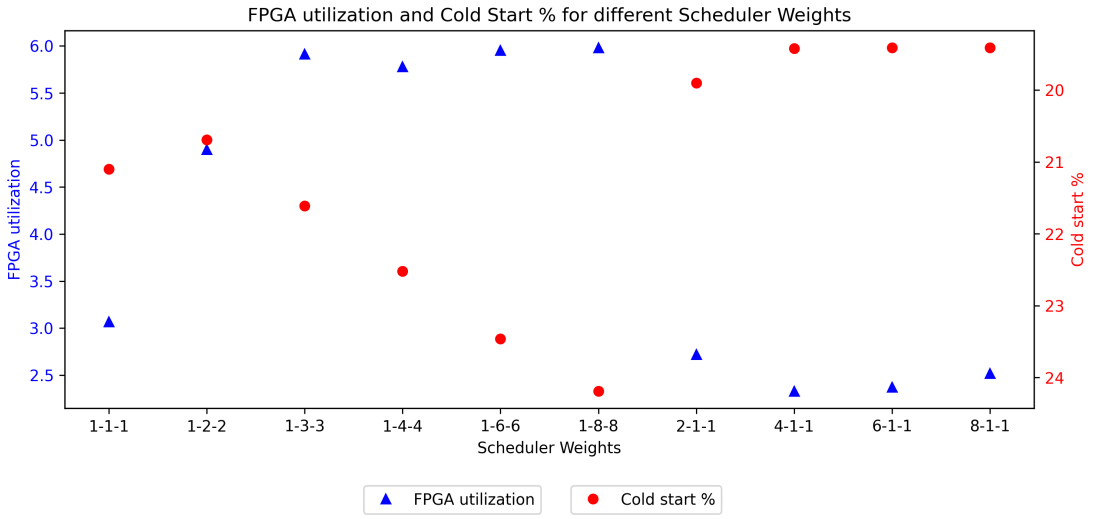


Figure 6.7: Effect of scheduler weights on FPGA utilization and cold start percentage. The y-axis plotting the cold start percentage is inverted as lower values are better. Scheduler weights are formatted as `<bitstream locality weight>-<usage time weight>-<reconfiguration time weight>`

FPGA utilization of 2.33s and a cold start percentage of 19.42%. While the cold start percentage is about 6.14% lower than in the case of preferring FPGA usage (1-2-2), the median FPGA utilization is 52.45% lower as well. The Pareto improvement from 1-1-1 to 1-2-2 is the best available trade-off between the two goals, as increasing the weights further leads to a higher cold start percentage or lower FPGA utilization respectively. For the following experiments, we will use scheduler weights of 1-2-2.

6.8 Simulator Assumptions

The simulator is built to assess the impact of placement decisions on resource utilization and fairness. Because of this, the current implementation does not take into account secondary factors such as network latency. We also do not evaluate concurrent request arrival as preemption of FPGA resources is not assumed for the scope of this project. As the underlying traces are not originating from hardware-accelerated functions, there are no existing data points on the distribution of FPGA usage compared to CPU usage for individual applications, therefore we assume a fixed ratio spent on each compute resource per request, creating both a baseline compute utilization as well as an FPGA-specific resource utilization (See Section 6.5). Lastly, we do not capture

changes occurring between a scheduling decision and function startup on a node. Furthermore, time only advances between function invocations, so system utilization is measured exclusively when an invocation is received as opposed to idle times, a necessary simplification to allow for the efficient simulation of large workloads.

6.9 Results

6.9.1 Utilization-based Function Placement

Using available utilization metrics allows for more equal distribution of functions across nodes in terms of real resource usage. We expect a higher per-node utilization, which should lead to lower per-node FPGA reconfigurations. Furthermore, we expect a lower overall cold start percentage, as equally-distributed functions should use the limited slots more efficiently. Related work has been done in [57] to extend the Kubernetes scheduler to consider CPU and memory utilization metrics for workload placement.

	Baseline	FPGA-aware	Difference	Difference in %
count	671.00	671.00	0.00	0.00
mean	26.70	30.78	4.08	15.26
std	178.10	177.57	-0.53	-0.30
min	0.00	0.00	0.00	NaN
p25	0.23	0.60	0.37	156.38
median	**3.18**	**4.58**	**1.40**	**44.03**
p75	16.72	22.67	5.95	35.62
p95	78.82	90.63	11.81	14.98
p99	351.02	323.18	-27.84	-7.93
max	4295.72	4254.12	-41.59	-0.97

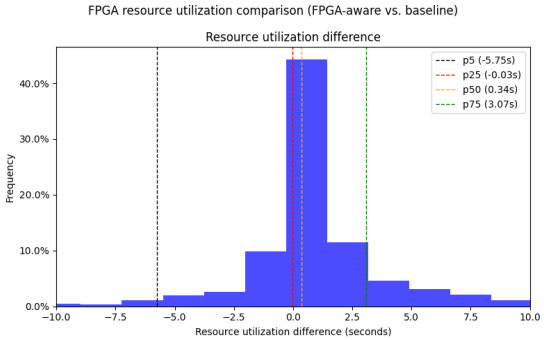
Table 6.1: Distribution of median system FPGA usage time (in seconds) over time for baseline vs. FPGA-aware scheduling.

To evaluate the impact of FPGA-aware scheduling on FPGA utilization, we perform two simulator runs with FPGA-aware scheduling enabled (weights of 1-2-2) compared to a disabled baseline (0-0-0) which only considers CPU utilization. Every 30s, we track CPU and FPGA utilization across all nodes and aggregate these values into the median system utilization over time.

In Table 6.1 we observe an increase in the overall median FPGA resource utilization from 3.18s to 4.58s (44.03%), leading to fewer wasted compute resources, higher energy efficiency, and higher downstream fairness due to equal workload distribution. In 95% of recorded median utilizations, the utilization is smaller than 323.18s, while the baseline value is 7.93% higher at 351.02s. This is largely due to the distributive effect of FPGA-aware scheduling, leading to fewer outlier nodes. At the same time, 25% of

recorded median FPGA utilization snapshots show values of up to 0.60s compared to 0.23s in the baseline system, a 156.38% increase of the lower boundary.

In addition to the overall median utilization, in Figure 6.8 we illustrate and compare the distribution of differences in utilization over time (subtracting the baseline utilization from FPGA-aware utilization for each data point to retrieve performance gains or losses), understanding points in time when FPGA awareness helped or impeded the scheduling performance. In the majority of cases, the difference is positive, indicating that the FPGA-aware scheduler improves the overall FPGA utilization of the system.



	Difference in FPGA usage time
count	671.00
mean	4.08
std	15.02
min	-54.01
25%	-0.03
50%	0.35
75%	3.07
max	119.23

Figure 6.8: Differences in FPGA usage times between baseline and FPGA-aware scheduling

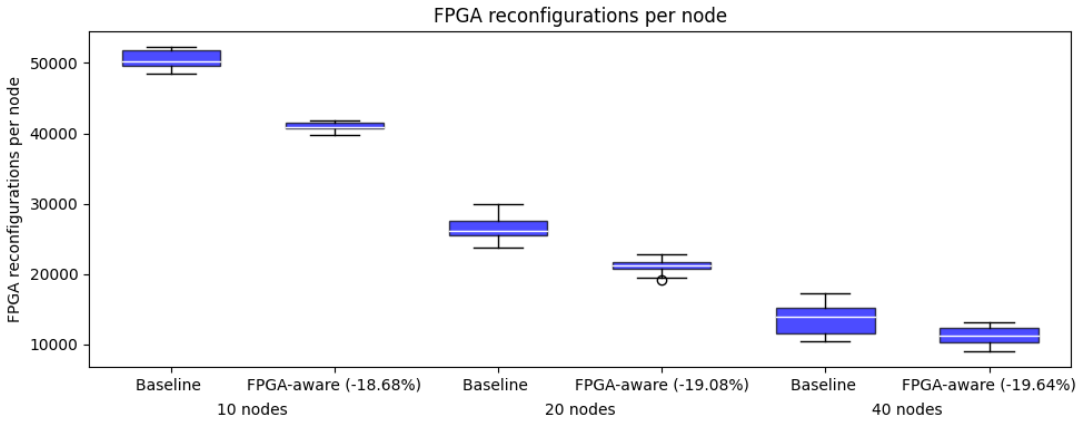


Figure 6.9: Comparison of reconfigurations per node for baseline and FPGA-aware scheduling

Furthermore, we see that increased distribution among nodes leads to lower per-node FPGA reconfigurations. In Figure 6.9, we visualize the difference in median FPGA

reconfigurations per node for different node counts and scheduler weights. With equal weights, so incorporating FPGA utilization metrics, we observe improvements up to 18.68% for ten nodes (See Table 6.2). This is in contrast to a fixed placement of functions which leads to high reconfigurations isolated to a few nodes, which causes contention on the limited FPGA slots.

In addition to decreased per-node FPGA reconfigurations, the overall cold start percentage is reduced by up to 18.89% for ten nodes. This trend in cold start reduction persists across different node counts, amounting to 18.89% for 10 nodes, 19.92% for 20 nodes, and 18.03% for 40 nodes respectively. Increasing the equal distribution of functions helps use existing compute resources more efficiently, and potentially decreases pressure on other resource types (CPU, memory), but as FPGA slots persist bitstreams after function invocations to avoid reconfiguration overhead in subsequent invocations, incoming requests that require different bitstreams still lead to cold starts, which is why we observe a non-zero cold start percentage.

	FPGA reconfigurations per node			Cold start %		
Nodes	Baseline	FPGA-aware	Δ	Baseline	FPGA-aware	Δ
10	50293.50	40899.50	-18.68%	25.51%	20.69%	-18.89%
20	26146.00	21158.50	-19.08%	26.71%	21.39%	-19.92%
40	13899.50	11169.00	-19.64%	27.57%	22.60%	-18.03%

Table 6.2: Benchmark results for p=0.5, m=0.01 (10ms), r=0.75, scheduler weights 0-0-0 (baseline) and 1-2-2 (FPGA-aware)

6.9.2 System Fairness

Dispersion of FPGA utilization	Baseline	FPGA-aware	Δ
cov(usage time)	1.35	1.07	-21.24%
cov(reconfiguration time)	0.94	0.52	-44.69%

Table 6.3: FPGA utilization dispersion for p=0.5, m=0.01 (10ms), r=0.75, scheduler weights 0-0-0 (baseline) and 1-2-2 (FPGA-aware)

Fairness is an important metric for FaaS, as it ensures that all tenants receive the same level of service. While no direct metric exists, we can interpret fairness as a function of the variability of per-node FPGA utilization. The higher the inequality of

FPGA utilization across nodes, the higher the risk of noisy neighbor effects, leading to performance degradation and lower fairness. Improving the distribution of function across nodes based on FPGA utilization metrics balances the workload more equally, leading to a lower risk of noisy neighbor effects, effectively increasing fairness.

Using FPGA utilization as an input for the scheduling algorithm forces the scheduler to place functions on nodes with lower utilization, which leads to more equal distribution of functions across nodes. We illustrate the impact of FPGA-aware function placement in Table 6.3, which compares the variability of per-node FPGA usage and reconfiguration times over time. The unitless coefficient of variation (CV) is calculated as the ratio of the standard deviation to the mean. A lower CV indicates a more equal distribution of FPGA usage across nodes, which is desirable for higher fairness. We observe a 21.24% and 44.69% decrease in the CV for FPGA utilization and reconfiguration times respectively, indicating a positive impact of FPGA-aware scheduling on fairness.

6.9.3 Conclusion

Keep in mind that the results presented in this section are based on assumptions on heterogeneous workloads in a serverless platform with FPGA acceleration capabilities. While we used real-world production traces from Microsoft Azure [72], we were required to make reasonable assumptions on the FPGA usage parameters of functions (See Section 6.5), as no real-world data exists for this purpose. However, we believe that the results presented in this section are a good indicator of the potential of FPGA-aware scheduling for serverless platforms with FPGA acceleration capabilities.

By incorporating FPGA utilization metrics into the scheduling algorithm, we can improve the overall utilization of FPGA resources, leading to lower FPGA reconfigurations per node and lower cold start percentages. Furthermore, we can increase fairness by distributing functions more equally across nodes. This is especially important for offering FPGA resources to multiple tenants, as existing FPGA resources can be used more efficiently, saving costs while offering the same performance. Running multiple simulation scenarios for ten nodes, scheduler weights preferring recent FPGA usage (1-2-2), and function FPGA usage parameters determined in Section 6.5, we observe improvements in the median FPGA utilization (increasing by up to 44.03% from 3.18s to 4.58s), FPGA reconfigurations per node (decreasing by up to 18.68%), and cold start percentages (decreasing by up to 18.89%), which indicate a positive impact of FPGA utilization metrics for scheduling serverless workloads on the system performance. Lastly, the variability in FPGA utilization across nodes is reduced by up to 21.24%, indicating a more equal utilization of FPGA resources and thus a positive effect on fairness.

7 Related Work

Related work can roughly be broken down into five major categories: FPGAs in the cloud, serverless scheduling and resource management, serverless workload characterization, FPGA architecture for multi-tenancy, and serverless FPGA sharing. Our work builds on existing research in all of these areas and extends it to enable the efficient use of FPGA resources in a multi-tenant serverless environment, achieving higher resource utilization and fairness in FPGA-accelerated serverless workloads compared to the baseline scheduler implementation.

FPGAs in the cloud enable developers to accelerate their applications with hardware without having to buy and maintain the hardware themselves. Byma, Steffan, Bannazadeh, et al. present virtualized FPGA-based hardware accelerators integrated with OpenStack and using partial reconfiguration [12]. Asiatici, George, Vipin, et al. design a runtime providing memory management, virtualization, and a hardware abstraction layer to enable developers to write FPGA-accelerated applications deployed on shared FPGAs [3]. Iordache, Pierre, Sanders, et al. propose FPGA groups, aggregating multiple FPGAs into a single virtual resource offered to multiple tenants [28]. Similarly, FPGAPooling [74] attempts to divide physical FPGA devices into multiple virtual slots. VineTalk [45] and ViTAL [70] are frameworks to enable the transparent use of shared FPGAs by offering a hardware abstraction layer to developers.

Cluster scheduling seeks to increase the scalability and fairness of batch jobs and other workloads by considering the orchestration of tasks. ERA [8] and Stratus [15] are cluster schedulers focusing on dollar-cost optimization of batch job execution in a dynamically-scalable environment. TetriSched [67] leverages runtime and deadline information to make global allocation decisions, reacting to changing cluster conditions as needed. Paragon [19] and Quasar [20] are cluster management systems classifying workloads and enabling users to specify performance constraints to achieve a higher resource utilization. Pace, Milios, Carra, et al. consider monitoring resource utilization for demand forecasting to improve cluster utilization [49].

Serverless scheduling and resource management attempts to increase the performance of serverless applications by improving function placement. Sequoia [66] is a quality-of-service (QoS) framework exposing flexible scheduling policies to developers and administrators. Hoseinyfarahabady, Y. C. Lee, Zomaya, and Tari propose a QoS-aware allocation controller to predict the dynamic runtime behavior of Lambda

functions and achieve QoS-aware resource allocation [26]. FaaSRank [69] and FaaSched [50] are function schedulers based on information monitored from servers and functions, learning scheduling policies through experience using reinforcement learning (RL). FnSched [65] and ENSURE [64] are function-level schedulers classifying incoming requests and concentrating load to minimize provider resource costs while meeting performance requirements. PASch [5] is a package-aware scheduling algorithm designed for functions depending on large packages or libraries. Kaffes, Yadwadkar, and Kozyrakis propose a cluster-level scheduler for serverless functions to improve elasticity and reduce interference, benefitting from a global view of cluster resources and direct assignment of functions to available cores [29]. Silva, Fireman, and Pereira evaluate starting functions from previously executed snapshots to improve the start-up time [62] of serverless functions. COCOA [23] is a cold start aware sizing method using queueing models to predict the required system capacity. Zuk and Rzadca leverage the composition of functions and installing environments in advance to significantly decrease response latency [75]. Fifer [25] is a framework for function chains, using utilization-aware bin packing for batching jobs and achieving SLO compliance by proactively spawning containers. Faa$T [56] is a transparent auto-scaling distributed cache for serverless applications, pre-warming the cache with objects likely to be accessed.

Serverless workload characterization attempts to understand real-world server workloads as a foundation to improve scheduling policies and system design. Shahrad, Fonseca, Goiri, et al. characterize production FaaS workloads and propose a hybrid-histogram policy predicting the next function invocation to enhance existing keep-alive policies for reducing cold starts [60]. Y. Zhang, Goiri, Chaudhry, et al. provide updated function invocation traces and propose hosting serverless FaaS platforms on harvested resources, VMs growing and shrinking based on unused compute resources [72]. Gsight [73] is an incremental learning model to predict workload performance under partial interference, improving function density by 18.79% while guaranteeing QoS.

FPGA architecture for multi-tenancy is a key enabler for serverless FaaS platforms to provide hardware acceleration as a service. Ringlein, Abel, Ditter, et al. present a system architecture managing network-attached FPGAs using partial reconfiguration, enabling scalable and agile FPGA management [55]. Coyote [35] is an abstraction layer enabling space-sharing using virtual FPGAs and partial reconfiguration.

Serverless FPGA sharing brings all the architectural and scheduling work together to provide serverless platforms with FPGA acceleration. BlastFunction [9] is a distributed FPGA sharing system for the acceleration of microservices and serverless applications in cloud environments. Mantle [54] is a system architecture abstracting device-specific FPGA logic, enabling scalable and portable FaaS offerings on disaggregated FPGAs. In [44], the authors explore a serverless deployment platform integrated directly on the FPGA board, using partial reconfiguration for multi-tenancy.

Existing work has focused on the individual disciplines of enabling FPGAs for the cloud, optimizing serverless function scheduling and delivery, creating FPGA architectures for multi-tenant and serverless use cases, and nascent efforts on making serverless platforms FPGA-aware. Creating a production-grade offering, however, requires the deep integration of these disciplines, combining FPGA architecture with an orchestration layer to produce a scalable and efficient serverless platform. Furthermore, workloads are expected to be heterogeneous by nature, with functions requiring different amounts of CPU, memory, and FPGA resources. Building systems on the notion of CPU-only or FPGA-only applications is not realistic given the diversity of workloads and the need for a general-purpose platform.

To our best knowledge, no public cloud FaaS provider or framework currently offers built-in serverless FPGA support, making this project the first to consider the FPGA-aware orchestration and scheduling of heterogeneous serverless functions in multi-tenant environments, based on industry-standard orchestration systems (Kubernetes) and FaaS frameworks (OpenFaaS). With our proposal, the FPGA becomes a first-class compute resource in the serverless platform, enabling more efficient use of the distributed FPGA hardware, higher system throughput, and reduced hotspots. This is a key building block in the creation of a large-scale serverless platform with FPGA acceleration, integrating with existing environments and providing a seamless experience for developers and administrators.

8 Summary

In this project, we present an extension to the Kubernetes scheduler to add FPGA awareness to a serverless platform, enabling the efficient use of FPGA resources in a multi-tenant environment. Based on the Kubernetes scheduling framework, the *fpgascheduling* plugin is compatible with existing Kubernetes deployments and can be used to enable FPGA acceleration support for existing serverless application platforms. Out of the box, we support defining FPGA vendor preferences for deployed workloads and tracking configured function bitstreams for enhanced scheduling. Furthermore, we extend the Funky Monitor unikernel hypervisor to collect workload-related metrics at runtime, which are then processed by the proposed metrics collector service running on each node and made available to the scheduler out of the critical path to ensure no performance degradation of scheduling decisions. To evaluate the system performance in a large-scale deployment without requiring extensive hardware and system stack configuration, we propose a simulator based on production traces from a real-world serverless platform [72]. Using the simulator, we can benchmark different scheduling strategies, and determine which data and criteria are useful for scheduling decisions.

In addition to existing criteria such as CPU utilization, scheduling decisions are now informed by real usage and system FPGA utilization, achieving a more uniform distribution of workloads. To the best of our knowledge, no existing data sets feature real-world estimates of time spent using hardware accelerators in serverless functions, so our experimental results retrieved from multiple simulator runs are informed by existing data (See Section 6.5) and show an increase in the overall median FPGA utilization by up to 44.03%, as well as decreasing FPGA reconfigurations per node by up to 18.68% to make efficient use of the existing hardware. Furthermore, the overall cold start percentage is reduced by up to 18.89%, leading to lower invocation latencies across the board. Most importantly, a more equal distribution of functions across nodes leads to higher fairness, measured by a decrease in the variability of FPGA usage times across nodes of 21.24%. Without the scheduler extension, FPGAs are either over or underutilized depending on the existing scheduler, leading to sub-optimal use of the available resources with significant financial and environmental costs.

The source code for the scheduler extension, the metrics collector, as well as the simulator is available on GitHub[1].

[1] `https://github.com/BrunoScheufler/serverless-fpga`

9 Future Work

While this project lays the groundwork necessary for integrating FPGAs into a serverless platform, there are still many untouched areas that can be explored in future work. In this chapter, we will discuss some of the most promising areas for future work.

We started reporting the most essential information related to FPGA utilization to the scheduler. Gathering further information about the system in the metrics collector and evaluating the impact using the simulator allows iteratively extending the scheduling criteria and improving the scheduler's performance over time. Providing additional information to the scheduler can be combined with efforts to increase FPGA resource sharing, such as using common kernels for shared functions such as (de-)serialization, encryption, and even providing application-specific tasks on the FPGA shared between functions, reducing the number of reconfigurations and increasing the utilization of accelerated functions. Adding awareness of the function kernel dependencies can inform placement decisions, especially when distinct kernels are available for different nodes. In environments with different FPGA device types, the scheduler can be extended to consider the device type when making scheduling decisions, improving the scalability in large heterogeneous environments. For the scope of this project, we assumed FPGA slots to be homogeneous, but in practice, different FPGA devices may offer a different number of slots, varying slot sizes for different bitstream sizes, or different performance characteristics. Ignoring this fact may lead to underutilization of the existing hardware and can be mitigated by providing the scheduler with information about the available FPGA slots and their characteristics, as well as tagging functions with the required slot size and performance characteristics. This information can be used to make better scheduling decisions and improve the utilization of the available resources.

Another interesting area is periodically de-provisioning functions to achieve a more uniform resource utilization, using Kubernetes de-scheduling [39]. As demands for functions change over time, the scheduler can be extended to de-provision functions that are no longer accessed, and provision them again when the demand increases, or move them to a different node to make better use of the available resources and stabilize the FPGA utilization. Especially the difference in invocation frequency observed in [72] makes for an interesting use case, as popular functions may only be detected after running for an extended period, and may be moved to another available node with

lower resource pressure to improve the performance of the function. After deployment to the most suitable node at that time, the system should measure the popularity of apps and after a while, converge to determine popular apps to distribute apart to reduce congestion affecting reconfigurations and performance. In the best case, we expect few popular apps with frequent invocations and a majority of apps with infrequent invocations placed on the same node, so that the popular apps can be served with low latency and the infrequent apps can be served just in time, accepting a minor reconfiguration delay.

While this project has primarily considered function placement as a lever for resource utilization and fairness in the system, there are potential gains in exploring runtime-related areas such as request routing and mediating FPGA access among multiple functions. Incoming requests are currently routed to any replica of a function, independent of the FPGA utilization of the node. By routing requests to the replica with the lowest FPGA utilization, we can further improve the utilization and fairness of the available resources. We can also reduce cold starts by routing requests to functions with bitstreams that are already loaded on the FPGA, while optionally warming up another FPGA slot on another replica if a high invocation frequency is detected. Furthermore, exploring different methods of queuing requests to access an FPGA slot to group requests from the same function and therefore reduce the number of reconfigurations can improve the performance of the system.

Lastly, tracking invocation frequency and building a model to predict the next invocation as in [60][69][50] allows pre-warming FPGAs just before the next request arrives, in which case the function can be served immediately without waiting for the FPGA to be reconfigured. This can be combined with the deprovisioning to pre-warm FPGAs for functions that are expected to be invoked shortly, and deprovision functions that are not expected to be invoked anytime soon to make better use of the available resources.

10 Appendix

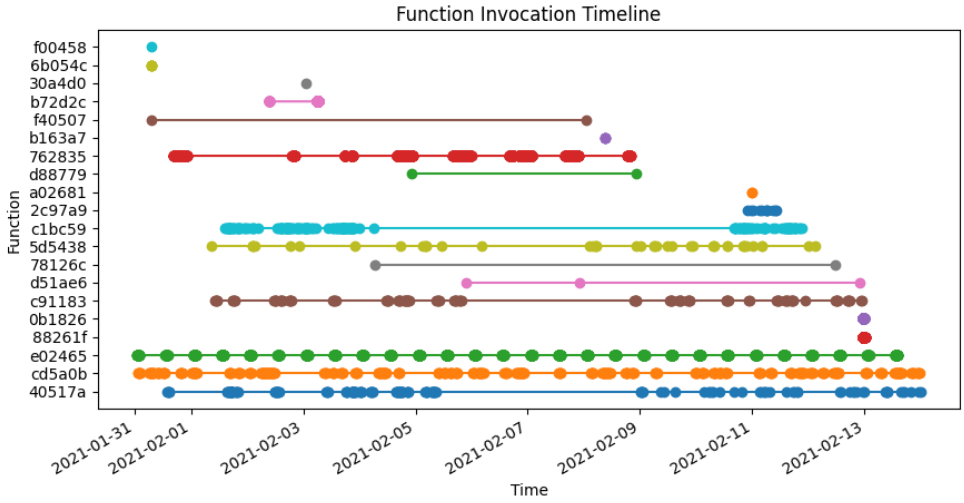
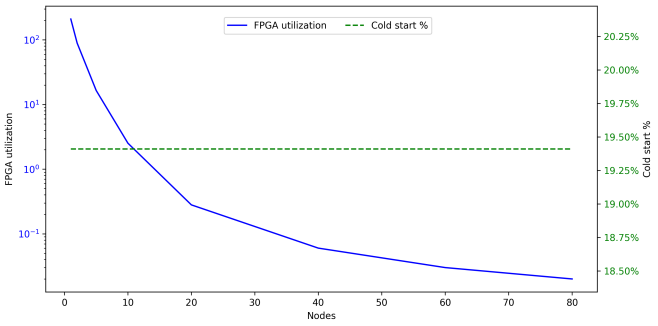


Figure 10.1: Invocation timeline of a sample of 20 functions from [72].



Nodes	FPGA utilization	Cold start %
1	210.01	19.41
2	89.14	19.41
5	16.49	19.41
10	2.52	19.41
20	0.28	19.41
40	0.06	19.41
60	0.03	19.41
80	0.02	19.41

Figure 10.2: Effect of number of nodes on median FPGA utilization and cold start percentage for scheduling weights 8-1-1

Glossary

bitstream A bitstream is an encoded binary file that contains the configuration data (logic gates, routing, etc.) for an FPGA. 1, 3, 6, 7, 21, 28, 41

bitstream locality A scheduling policy that prioritizes FPGAs with bitstreams already configured. 6, 13, 15, 19, 32

Container Runtime Interface (CRI) An interface between Kubernetes and container runtimes that makes it possible to plug any container runtime into the Kubernetes system. 11, 12, 17

Discrete-event simulation (DES) A type of simulation where the state of the system only changes at discrete points in time, as opposed to continuous simulation where the state variables change continuously over time. 26

Field-programmable gate array (FPGA) An integrated circuit designed to be configured by a customer or a designer after manufacturing. iv, 1–4, 6, 7, 9, 10, 13, 17–19, 21, 25, 31, 41, 45

Function-as-a-Service (FaaS) A category of cloud computing services that provides a platform allowing customers to develop, run, and manage application functionalities without the complexity of building and maintaining the infrastructure typically associated with developing and launching an app. iv, 1, 5, 10, 36, 46

Funky Monitor A hypervisor extended with hypercalls to acquire, use, and release FPGA slots. 3, 8, 10–12, 14, 18, 19, 21, 29, 41, 45, 46

hypercall Similar to a system call, a hypercall is a software trap from a guest virtual machine to the hypervisor or virtual machine monitor. 10, 11, 14, 17, 18, 21, 29

Hypervisor A piece of computer software, firmware or hardware that creates and runs virtual machines. 3, 14, 21, 41

kata-urunc A container runtime for Kata Containers that uses Funky Monitor. 8, 11, 12, 17

Kubernetes A portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation [37]. iv, 3, 7, 9, 11–13, 20, 41

Kubernetes annotations Annotations are key-value pairs that can be attached to Kubernetes objects and may contain arbitrary data. 13, 15, 46

metrics collector A system component that accepts metrics from the Funky Monitor metrics reporter, aggregates them, and stores them as Kubernetes annotations. 3, 10, 13, 14, 20, 22, 41

multi-tenant A software architecture in which a single instance of a software serves multiple tenants (customers). iv, 3, 5, 6, 10, 41

noisy neighbor effects A situation where a tenant's performance is negatively affected by the resource usage of other tenants. 5, 31, 37

Open Container Initiative (OCI) Image A specification for container images, which include both a filesystem bundle and configuration metadata for the container. 17, 22

OpenCL A framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors [48]. 6, 7, 17–19, 21

OpenFaaS An open-source serverless platform built on top of Kubernetes. 1, 8, 9, 11, 12, 14, 17

production traces Data set of real-world traces of FaaS workloads containing invocation timestamps, durations, and function identifiers [72]. iv, 4, 26, 41

Shell (FPGA) A static region on an FPGA that provides a fixed set of resources for core logic and I/O. 6, 7, 14, 19

Unikernel A specialized, single-address-space machine image constructed by using library operating systems [43]. 3, 7, 8, 10–12, 14, 17, 19, 21, 41

Xilinx Runtime (XRT) A software stack that provides drivers and libraries for FPGA acceleration boards from Xilinx. 7, 9, 17–19, 21

List of Figures

Bibliography

[1] *Amazon EC2 F1 Instances*. Amazon Web Services, Inc. URL: https://aws.amazon.com/ec2/instance-types/f1/ (visited on 05/26/2023).

[2] S. Asano, T. Maruyama, and Y. Yamaguchi. "Performance comparison of FPGA, GPU and CPU in image processing." In: *2009 International Conference on Field Programmable Logic and Applications*. 2009 International Conference on Field Programmable Logic and Applications. ISSN: 1946-1488. Aug. 2009, pp. 126–131. DOI: 10.1109/FPL.2009.5272532.

[3] 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)*. 2016 26th International Conference on Field Programmable Logic and Applications (FPL). ISSN: 1946-1488. Aug. 2016, pp. 1–2. DOI: 10.1109/FPL.2016.7577389.

[4] 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 IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW). Phoenix, AZ, USA: IEEE, May 2014, pp. 228–235. ISBN: 978-1-4799-4116-2 978-1-4799-4117-9. DOI: 10.1109/IPDPSW.2014.30.

[5] G. Aumala, E. Boza, L. Ortiz-Avilés, G. Totoy, and C. Abad. "Beyond Load Balancing: Package-Aware Scheduling for Serverless Platforms." In: *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID). May 2019, pp. 282–291. DOI: 10.1109/CCGRID.2019.00042.

[6] *AWS Lambda*. Amazon Web Services, Inc. URL: https://aws.amazon.com/lambda/ (visited on 07/10/2023).

[7] *Azure Functions*. URL: https://azure.microsoft.com/en-us/products/functions (visited on 07/10/2023).

[8] M. Babaioff, Y. Mansour, N. Nisan, G. Noti, C. Curino, N. Ganapathy, I. Menache, O. Reingold, M. Tennenholtz, and E. Timnat. "ERA: A Framework for Economic Resource Allocation for the Cloud." In: *Proceedings of the 26th International Conference on World Wide Web Companion - WWW '17 Companion*. 2017, pp. 635–642. DOI: 10.1145/3041021.3054186. arXiv: 1702.07311[cs].

[9] M. Bacis, R. Brondolin, and M. D. Santambrogio. "BlastFunction: an FPGA-as-a-Service system for Accelerated Serverless Computing." In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE). ISSN: 1558-1101. Mar. 2020, pp. 852–857. DOI: 10.23919/DATE48585.2020.9116333.

[10] L. A. Barroso. "Warehouse-Scale Computing: Entering the Teenage Decade." In: *Proceedings of the 38th annual international symposium on Computer architecture*. ISCA '11. New York, NY, USA: Association for Computing Machinery, June 6, 2011. ISBN: 978-1-4503-0472-6. DOI: 10.1145/2000064.2019527.

[11] 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)*. 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom). Vancouver, BC: IEEE, Nov. 2015, pp. 250–257. ISBN: 978-1-4673-9560-1. DOI: 10.1109/CloudCom.2015.89.

[12] S. Byma, J. G. Steffan, H. Bannazadeh, A. Leon-Garcia, and P. Chow. "FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack." In: *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. 2014 IEEE 22nd International Symposium on Field-Programmable Custom Computing Machines (FCCM 2014). Boston, MA, USA: IEEE, May 2014, pp. 109–116. ISBN: 978-1-4799-5111-6. DOI: 10.1109/FCCM.2014.42.

[13] J. Casper and K. Olukotun. "Hardware acceleration of database operations." In: *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*. FPGA'14: The 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. Monterey California USA: ACM, Feb. 26, 2014, pp. 151–160. ISBN: 978-1-4503-2671-1. DOI: 10.1145/2554688.2554787.

[14] 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: (Oct. 2016).

[15] A. Chung, J. W. Park, and G. R. Ganger. "Stratus: cost-aware container scheduling in the public cloud." In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '18: ACM Symposium on Cloud Computing. Carlsbad CA USA: ACM, Oct. 11, 2018, pp. 121–134. ISBN: 978-1-4503-6011-1. DOI: 10.1145/3267809.3267819.

[16] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. El Husseini, T. Juhasz, K. Kagi, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Yi Xiao, D. Zhang, R. Zhao, and D. Burger. "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave." In: *IEEE Micro* 38.2 (Mar. 2018). Conference Name: IEEE Micro, pp. 8–20. ISSN: 1937-4143. DOI: 10.1109/MM.2018.022071131.

[17] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. "Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms." In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17: ACM SIGOPS 26th Symposium on Operating Systems Principles. Shanghai China: ACM, Oct. 14, 2017, pp. 153–167. ISBN: 978-1-4503-5085-3. DOI: 10.1145/3132747.3132772.

[18] 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*. FPGA'16: The 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. Monterey California USA: ACM, Feb. 21, 2016, pp. 105–110. ISBN: 978-1-4503-3856-1. DOI: 10.1145/2847263.2847339.

[19] C. Delimitrou and C. Kozyrakis. "Paragon: QoS-aware scheduling for heterogeneous datacenters." In: *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*. ASPLOS '13. New York, NY, USA: Association for Computing Machinery, Mar. 16, 2013, pp. 77–88. ISBN: 978-1-4503-1870-9. DOI: 10.1145/2451116.2451125.

[20] C. Delimitrou and C. Kozyrakis. "Quasar: resource-efficient and QoS-aware cluster management." In: *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. ASPLOS '14. New York, NY, USA: Association for Computing Machinery, Feb. 24, 2014, pp. 127–144. ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541941.

[21] L. Deng, Y.-L. Ren, F. Xu, H. He, and C. Li. "Resource Utilization Analysis of Alibaba Cloud." In: *Intelligent Computing Theories and Application*. Ed. by D.-S. Huang, V. Bevilacqua, P. Premaratne, and P. Gupta. Vol. 10954. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 183–194. ISBN: 978-3-319-95929-0 978-3-319-95930-6. DOI: 10.1007/978-3-319-95930-6_18.

[22] X. Fan, W.-D. Weber, and L. A. Barroso. "Power provisioning for a warehouse-sized computer." In: *Proceedings of the 34th annual international symposium on Computer architecture*. ISCA '07. New York, NY, USA: Association for Computing Machinery, June 9, 2007, pp. 13–23. ISBN: 978-1-59593-706-3. DOI: 10.1145/1250662.1250665.

[23] A. U. Gias and G. Casale. "COCOA: Cold Start Aware Capacity Planning for Function-as-a-Service Platforms." In: *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). ISSN: 2375-0227. Nov. 2020, pp. 1–8. DOI: 10.1109/MASCOTS50786.2020.9285966.

[24] *Google Cloud Functions*. Google Cloud. URL: https://cloud.google.com/functions (visited on 07/10/2023).

[25] J. R. Gunasekaran, P. Thinakaran, N. Chidambaram, M. T. Kandemir, and C. R. Das. *Fifer: Tackling Underutilization in the Serverless Era*. Aug. 28, 2020. DOI: 10.48550/arXiv.2008.12819. arXiv: 2008.12819[cs].

[26] Hoseinyfarahabady, Y. C. Lee, A. Y. Zomaya, and Z. Tari. "A QoS-Aware Resource Allocation Controller for Function as a Service (FaaS) Platform." In: *Service-Oriented Computing*. Ed. by M. Maximilien, A. Vallecillo, J. Wang, and M. Oriol. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 241–255. ISBN: 978-3-319-69035-3. DOI: 10.1007/978-3-319-69035-3_17.

[27] L. Huan. *Host server CPU utilization in Amazon EC2 cloud*. Huan Liu's Blog. Feb. 17, 2012. URL: https://huanliu.wordpress.com/2012/02/17/ (visited on 06/13/2023).

[28] A. Iordache, G. Pierre, P. Sanders, J. G. de F. Coutinho, and M. Stillwell. "High performance in the cloud with FPGA groups." In: *Proceedings of the 9th International Conference on Utility and Cloud Computing*. UCC '16. New York, NY, USA: Association for Computing Machinery, Dec. 6, 2016, pp. 1–10. ISBN: 978-1-4503-4616-0. DOI: 10.1145/2996890.2996895.

[29] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis. "Centralized Core-granular Scheduling for Serverless Functions." In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '19. New York, NY, USA: Association for Computing Machinery, Nov. 20, 2019, pp. 158–164. ISBN: 978-1-4503-6973-2. DOI: 10.1145/3357223.3362709.

[30] 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 26th International Conference on Field Programmable Logic and Applications (FPL). Lausanne, Switzerland: IEEE, Aug. 2016, pp. 1–4. ISBN: 978-2-8399-1844-2. DOI: 10.1109/FPL.2016.7577353.

[31] Kata Containers and contributors. *Kata Containers - a standard implementation of lightweight Virtual Machines (VMs) that feel and perform like containers, but provide the workload isolation and security advantages of VMs.* URL: https://github.com/kata-containers/kata-containers (visited on 06/23/2023).

[32] D. K. Kim and H.-G. Roh. "Scheduling Containers Rather Than Functions for Function-as-a-Service." In: *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid). May 2021, pp. 465–474. DOI: 10.1109/CCGrid51090.2021.00056.

[33] *Knative*. URL: https://knative.dev/docs/ (visited on 07/10/2023).

[34] O. Knodel, P. R. Genssler, and R. G. Spallek. "Migration of long-running Tasks between Reconfigurable Resources using Virtualization." In: *ACM SIGARCH Computer Architecture News* 44.4 (Jan. 11, 2017), pp. 56–61. ISSN: 0163-5964. DOI: 10.1145/3039902.3039913.

[35] 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). 2020, pp. 991–1010. ISBN: 978-1-939133-19-9.

[36] *kube-scheduler*. Kubernetes. Section: docs. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/ (visited on 05/29/2023).

[37] Kubernetes, Google, Inc., and contributors. *Kubernetes (K8s)*. original-date: 2014-06-06T22:56:04Z. June 23, 2023. URL: https://github.com/kubernetes/kubernetes (visited on 06/23/2023).

[38] Kubernetes Contributors. *community/scheduler_queues.md at master · kubernetes/-community · GitHub*. URL: https://github.com/kubernetes/community/blob/master/contributors/devel/sig-scheduling/scheduler_queues.md (visited on 05/29/2023).

[39] Kubernetes Contributors. *Descheduler for Kubernetes*. original-date: 2017-07-28T17:11:38Z. May 26, 2023. URL: https://github.com/kubernetes-sigs/descheduler (visited on 05/26/2023).

[40] 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). 2020, pp. 955–971. ISBN: 978-1-939133-19-9.

[41] J. Leverich and C. Kozyrakis. "On the energy (in)efficiency of Hadoop clusters." In: *ACM SIGOPS Operating Systems Review* 44.1 (Mar. 12, 2010), pp. 61–65. ISSN: 0163-5980. DOI: 10.1145/1740390.1740405.

[42] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey. "Architecting to achieve a billion requests per second throughput on a single key-value store server platform." In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ISCA '15: The 42nd Annual International Symposium on Computer Architecture. Portland Oregon: ACM, June 13, 2015, pp. 476–488. ISBN: 978-1-4503-3402-0. DOI: 10.1145/2749469.2750416.

[43] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. "Unikernels: library operating systems for the cloud." In: *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*. ASPLOS '13. New York, NY, USA: Association for Computing Machinery, Mar. 16, 2013, pp. 461–472. ISBN: 978-1-4503-1870-9. DOI: 10.1145/2451116.2451167.

[44] F. Maschi, D. Korolija, and G. Alonso. "Serverless FPGA: Work-In-Progress." In: *Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies*. SESAME '23. New York, NY, USA: Association for Computing Machinery, May 8, 2023, pp. 1–4. ISBN: 9798400701856. DOI: 10.1145/3592533.3592804.

[45] 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)*. 2017 27th International Conference on Field Programmable Logic and Applications (FPL). ISSN: 1946-1488. Sept. 2017, pp. 1–4. DOI: 10.23919/FPL.2017.8056788.

[46] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. "Power management of online data-intensive services." In: *ACM SIGARCH Computer Architecture News* 39.3 (June 4, 2011), pp. 319–330. ISSN: 0163-5964. DOI: 10.1145/2024723.2000103.

[47] 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*. FPGA'16: The 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. Monterey California USA: ACM, Feb. 21, 2016, pp. 111–117. ISBN: 978-1-4503-3856-1. DOI: 10.1145/2847263.2847337.

[48] *OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems*. The Khronos Group. Section: API. July 21, 2013. URL: https://www.khronos.org// (visited on 06/23/2023).

[49] F. Pace, D. Milios, D. Carra, D. Venzano, and P. Michiardi. *A Data-Driven Approach to Dynamically Adjust Resource Allocation for Compute Clusters*. July 1, 2018. DOI: 10.48550/arXiv.1807.00368. arXiv: 1807.00368[cs].

[50] A. Panda and S. R. Sarangi. *FaaSched: A Jitter-Aware Serverless Scheduler*. Mar. 11, 2023. DOI: 10.48550/arXiv.2303.06473. arXiv: 2303.06473[cs].

[51] P. Papaphilippou and W. Luk. "Accelerating Database Systems Using FPGAs: A Survey." In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 2018 28th International Conference on Field Programmable Logic and Applications (FPL). ISSN: 1946-1488. Aug. 2018, pp. 125–1255. DOI: 10.1109/FPL.2018.00030.

[52] 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 (May 2015). Conference Name: IEEE Micro, pp. 10–22. ISSN: 1937-4143. DOI: 10.1109/MM.2015.42.

[53] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. "Heterogeneity and dynamicity of clouds at scale: Google trace analysis." In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. New York, NY, USA: Association for Computing Machinery, Oct. 14, 2012, pp. 1–13. ISBN: 978-1-4503-1761-0. DOI: 10.1145/2391229.2391236.

[54] B. Ringlein, F. Abel, D. Diamantopoulos, B. Weiss, C. Hagleitner, M. Reichenbach, and D. Fey. "A Case for Function-as-a-Service with Disaggregated FPGAs." In: *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. 2021 IEEE 14th International Conference on Cloud Computing (CLOUD). Chicago, IL, USA: IEEE, Sept. 2021, pp. 333–344. ISBN: 978-1-66540-060-2. DOI: 10.1109/CLOUD53861.2021.00047.

[55] B. Ringlein, F. Abel, A. Ditter, B. Weiss, C. Hagleitner, and D. Fey. "System Architecture for Network-Attached FPGAs in the Cloud using Partial Reconfiguration." In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 2019 29th International Conference on Field Programmable Logic and Applications (FPL). ISSN: 1946-1488. Sept. 2019, pp. 293–300. DOI: 10.1109/FPL.2019.00054.

[56] F. Romero, G. I. Chaudhry, Í. Goiri, P. Gopa, P. Batum, N. J. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini. "Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications." In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '21: ACM Symposium on Cloud Computing. Seattle WA USA: ACM, Nov. 2021, pp. 122–137. ISBN: 978-1-4503-8638-8. DOI: 10.1145/3472883.3486974.

[57] *scheduler-plugins/kep/61-Trimaran-real-load-aware-scheduling at master · kubernetes-sigs/scheduler-plugins*. GitHub. URL: https://github.com/kubernetes-sigs/scheduler-plugins/tree/master/kep/61-Trimaran-real-load-aware-scheduling (visited on 05/07/2023).

[58] *Scheduling Framework*. Kubernetes. Section: docs. URL: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/ (visited on 05/26/2023).

[59] S. Sebastio, K. S. Trivedi, and J. Alonso. "Characterizing machines lifecycle in Google data centers." In: *Performance Evaluation* 126 (2018), pp. 39–63. ISSN: 0166-5316. DOI: https://doi.org/10.1016/j.peva.2018.08.001.

[60] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini. *Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider*. June 5, 2020. DOI: 10.48550/arXiv.2003.03423. arXiv: 2003.03423[cs].

[61] 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 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). Taipei, Taiwan: IEEE, Oct. 2016, pp. 1–12. ISBN: 978-1-5090-3508-3. DOI: 10.1109/MICRO.2016.7783720.

[62] P. Silva, D. Fireman, and T. E. Pereira. "Prebaking Functions to Warm the Serverless Cold Start." In: *Proceedings of the 21st International Middleware Conference*. Middleware '20. New York, NY, USA: Association for Computing Machinery, Dec. 11, 2020, pp. 1–13. ISBN: 978-1-4503-8153-6. DOI: 10.1145/3423211.3425682.

[63] 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*. FPGA'16: The 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. Monterey California USA: ACM, Feb. 21, 2016, pp. 16–25. ISBN: 978-1-4503-3856-1. DOI: 10.1145/2847263.2847276.

[64] A. Suresh, G. Somashekar, A. Varadarajan, V. R. Kakarla, H. Upadhyay, and A. Gandhi. "ENSURE: Efficient Scheduling and Autonomous Resource Management in Serverless Environments." In: *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS). Aug. 2020, pp. 1–10. DOI: 10.1109/ACSOS49614.2020.00020.

[65] A. Suresh and A. Gandhi. "FnSched: An Efficient Scheduler for Serverless Functions." In: *Proceedings of the 5th International Workshop on Serverless Computing*. WOSC '19. New York, NY, USA: Association for Computing Machinery, Dec. 9, 2019, pp. 19–24. ISBN: 978-1-4503-7038-7. DOI: 10.1145/3366623.3368136.

[66] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka. "Sequoia: enabling quality-of-service in serverless computing." In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. SoCC '20. New York, NY, USA: Association for Computing Machinery, Oct. 12, 2020, pp. 311–327. ISBN: 978-1-4503-8137-6. DOI: 10.1145/3419111.3421306.

[67] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. "TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters." In: *Proceedings of the Eleventh European Conference on Computer Systems*. EuroSys '16. New York, NY, USA: Association for Computing Machinery, Apr. 18, 2016, pp. 1–16. ISBN: 978-1-4503-4240-7. DOI: 10.1145/2901318.2901355.

[68] D. Wang, K. Xu, and D. Jiang. "PipeCNN: An OpenCL-based open-source FPGA accelerator for convolution neural networks." In: *2017 International Conference on Field Programmable Technology (ICFPT)*. 2017 International Conference on Field Programmable Technology (ICFPT). Dec. 2017, pp. 279–282. DOI: 10.1109/FPT.2017.8280160.

[69] H. Yu, A. A. Irissappane, H. Wang, and W. J. Lloyd. "FaaSRank: Learning to Schedule Functions in Serverless Platforms." In: *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. 2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS). Washington, DC, USA: IEEE, Sept. 2021, pp. 31–40. ISBN: 978-1-66541-261-2. DOI: 10.1109/ACSOS52086.2021.00023.

[70] Y. Zha and J. Li. "Virtualizing FPGAs in the Cloud." In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20: Architectural Support for Programming Languages and Operating Systems. Lausanne Switzerland: ACM, Mar. 9, 2020, pp. 845–858. ISBN: 978-1-4503-7102-5. DOI: 10.1145/3373376.3378491.

[71] 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*. FPGA '15: The 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. Monterey California USA: ACM, Feb. 22, 2015, pp. 161–170. ISBN: 978-1-4503-3315-3. DOI: 10.1145/2684746.2689060.

[72] Y. Zhang, Í. Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini. "Faster and Cheaper Serverless Computing on Harvested Resources." In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*. SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles. Virtual Event Germany: ACM, Oct. 26, 2021, pp. 724–739. ISBN: 978-1-4503-8709-5. DOI: 10.1145/3477132.3483580.

[73] L. Zhao, Y. Yang, Y. Li, X. Zhou, and K. Li. "Understanding, predicting and scheduling serverless workloads under partial interference." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '21. New York, NY, USA: Association for Computing Machinery, Nov. 13, 2021, pp. 1–15. ISBN: 978-1-4503-8442-1. DOI: 10.1145/3458817.3476215.

[74] Z. Zhu, A. X. Liu, F. Zhang, and F. Chen. "FPGA Resource Pooling in Cloud Computing." In: *IEEE Transactions on Cloud Computing* 9.2 (Apr. 2021). Conference Name: IEEE Transactions on Cloud Computing, pp. 610–626. ISSN: 2168-7161. DOI: 10.1109/TCC.2018.2874011.

[75] P. Zuk and K. Rzadca. "Scheduling Methods to Reduce Response Latency of Function as a Service." In: *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing

(SBAC-PAD). Porto, Portugal: IEEE, Sept. 2020, pp. 132–140. ISBN: 978-1-72819-924-5. DOI: 10.1109/SBAC-PAD49847.2020.00028.