Funky: Cloud-Native FPGA Virtualization and Orchestration

Atsushi Koshiba
Technical University of Munich
Munich, Germany
atsushi.koshiba@tum.de

Charalampos Mainas
Technical University of Munich
Munich, Germany
charalampos.mainas@tum.de

Pramod Bhatotia
Technical University of Munich
Munich, Germany
pramod.bhatotia@tum.de

Abstract

The adoption of FPGAs in cloud-native environments is facing impediments due to FPGA limitations and CPU-oriented design of orchestrators, as they lack virtualization, isolation, and preemption support for FPGAs. Consequently, cloud providers offer *no* orchestration services for FPGAs, leading to low scalability, flexibility, and resiliency.

This paper presents Funky, a full-stack FPGA-aware orchestration engine for cloud-native applications. Funky offers primary orchestration services for FPGA workloads to achieve high performance, utilization, scalability, and fault tolerance, accomplished by three contributions: (1) FPGA virtualization for lightweight sandboxes, (2) FPGA state management enabling task preemption and checkpointing, and (3) FPGA-aware orchestration components following the industry-standard CRI/OCI specifications.

We implement and evaluate Funky using four x86 servers with Alveo U50 FPGA cards. Our evaluation highlights that Funky allows us to port 23 OpenCL applications from the Xilinx Vitis and Rosetta benchmark suites by modifying 3.4% of the source code while keeping the OCI image sizes 28.7× smaller than AMD's FPGA-accessible Docker containers. In addition, Funky incurs only 7.4% performance overheads compared to native execution, while providing virtualization support with strong hypervisor-enforced isolation and cloud-native orchestration for a set of distributed FPGAs. Lastly, we evaluate Funky's orchestration services in a large-scale cluster using Google production traces, showing its scalability, fault tolerance, and scheduling efficiency.

CCS Concepts

• **Computer systems organization → Cloud computing**.

Keywords

FPGA, hardware acceleration, microservices, cloud orchestration, FPGA virtualization

ACM Reference Format:
Atsushi Koshiba, Charalampos Mainas, and Pramod Bhatotia. 2025. Funky: Cloud-Native FPGA Virtualization and Orchestration. In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3772052.3772226

SoCC '25, November 19–21, 2025, Online, USA

ACM ISBN 979-8-4007-2276-9/2025/11
https://doi.org/10.1145/3772052.3772226

1 Introduction

Cloud-native architectures are a promising trend in the cloud, where various workloads designed as collections of small services, i.e., microservices [12, 59], are deployed across server nodes by an orchestrator (e.g., Kubernetes [73], Mesos [53], Swarm [45], YARN [52]) on behalf of users. The orchestrators play an important role in efficient cloud resource management, including resource provisioning, workload scaling [114], scheduling [23, 135, 137, 140], migration [15, 126, 129, 159], and fault tolerance [1, 120, 138].

In the meantime, hardware accelerators are rapidly adopted by major cloud providers, such as GPUs [29, 35], TPUs [62], and FPGAs [6, 28, 113], to meet the computational demands of modern cloud workloads. Despite their promising performance benefits, existing cloud orchestrators are natively designed to manage *only* CPU and memory resources [42] and lack comprehensive support for these accelerators. Because accelerators are more limited and expensive than CPUs due to a small number of PCIe slots per server node [6, 35, 103], orchestration support for accelerators is crucial to efficiently share them across millions of services running in a multi-tenant cloud.

Although various hardware accelerators are available in modern cloud environments, we primarily focus on FPGAs because of their flexibility and customizability. FPGAs are programmable hardware where users can configure custom logic specialized to a specific computation, making them attractive for ever-changing cloud workloads. As a result, FPGAs are widely offered by all major cloud providers [6, 28, 113], and are shown to be effective for machine learning [128, 162], databases [20], distributed applications [61], storage stack [90], and graph processing [8, 37, 109].

Despite FPGAs' promise to accelerate various cloud workloads in an energy-efficient manner, there is currently *no orchestration engine* for FPGAs [42, 115, 139]. We identify three key challenges that hinder FPGA orchestration, which are not fully addressed by existing industry practices [56, 152] and academic solutions [38, 47].

First, no lightweight sandboxes for FPGA applications exist. An execution sandbox (e.g., VM) is paramount to virtualizing and sharing cloud resources across multi-tenant applications. Cloud-native environments adopt lightweight sandboxes such as containers [132, 160] and lightweight VMs [4, 100, 118, 133] for small performance penalties [123] and low start-up latencies [48, 88]. However, FPGA virtualization on these sandboxes has not yet been explored. While FPGA vendors offer pre-built Docker containers [155], they not only lack FPGA virtualization but also need a large portion of FPGA system stacks installed, sacrificing their lightweight nature.

Second, FPGA-accelerated workloads are not preemptible/recoverable. Cloud-native environments deploy not only stateless workloads (e.g., serverless functions), to which prior studies [38, 47] adapt FPGAs, but also long-running stateful microservices [30, 110, 131] that keep alive and repeatedly handle incoming requests.

Stateful workloads are also primary targets for FPGA acceleration, such as databases [39, 51, 144], machine learning [122, 142], and search engines [113]. However, the lack of FPGA state management (e.g., context switch) complicates FPGA sharing across them. For instance, once such workloads get and initialize any FPGA, they want to keep it 'warmed up' to avoid paying high bootup costs again (e.g., FPGA reconfiguration, input data initialization) for low latencies [112]. Allowing long-running services to occupy FPGAs without preemption leads to severe resource underutilization and also data loss by system failures [138].

Third, existing orchestrators are unaware of FPGA resources. Existing cloud orchestrators offer limited FPGA support due to the lack of virtualization mechanisms. Device plugins [56, 152] do not support FPGA sharing across multi-tenant containers. Extending orchestrators for FPGA support is challenging because they follow open specifications such as Container Runtime Interface (CRI) [31] and Open Container Initiative (OCI) [33], which are not designed for accelerator management. Any extension must not violate these specifications to guarantee system compatibility.

To overcome them, we propose Funky, an FPGA orchestration engine for cloud-native applications. Funky is the first system that tackles an end-to-end orchestration support for cloud FPGAs, achieving three contributions:

(1) **Lightweight FPGA virtualization** (§ 3.2, § 3.3). We design a unikernel-based lightweight sandbox for FPGA virtualization. Unikernels [87, 89, 98, 100, 104] are specialized OSes designed to execute a single, specific application. Our unikernel is designed for FPGA applications, which encapsulates individual CPU applications using FPGAs and isolates them from underlying FPGA system stacks [54, 85, 157]. It is not only suitable for cloud-native applications but also allows the orchestrator to transparently allocate/deallocate FPGAs to tasks on demand.
(2) **FPGA state management** (§ 3.4). We design a task state management mechanism to suspend and resume FPGA workloads. It enables evicting, migrating, and checkpointing tasks whose states are distributed across CPU and FPGA devices. Unikernel's simplified state and single-process basis facilitate the underlying hypervisor in tracing FPGA I/O requests from each guest sandbox and saving/loading CPU and FPGA states.
(3) **FPGA-aware orchestration** (§ 3.5). We demonstrate an end-to-end design integration of Funky's virtualization and state management mechanisms into industry-standard orchestrators without violating the CRI/OCI specifications [31, 33]. It provides three primary FPGA orchestration services: preemptive scheduling, checkpointing, and workload scaling.

We implement the Funky framework for Alveo U50 FPGA with the Vitis Shell [157]. We extend IncludeOS [17] and Solo5 [147] to implement the unikernel sandbox and dedicated hypervisor. We also implement a prototype of our orchestrator and container runtime, offering high-level orchestration services for FPGA workloads in cooperation with the sandbox.

We evaluate Funky across two dimensions. First, we evaluate Funky's virtualization overheads, portability, and state management using two FPGA benchmark suites, i.e., Vitis benchmarks [153] and Rosetta's real-world applications [164], on a real four-node FPGA cluster. The results highlight that Funky allows us to port 23 OpenCL applications by modifying 3.4% of the source code with 28.7× smaller image size on average than Docker containers maintained by an FPGA vendor (AMD) [155]. Funky imposes performance overheads of 7.4% compared to the native execution, only 0.6% higher than AMD's containers that do not virtualize FPGAs. Second, we evaluate Funky using Google production traces [137] to showcase its orchestration effectiveness for a large-scale cluster.

2 Background and Motivation

2.1 FPGAs in Cloud Environments

This paper targets cloud-native environments, where applications consist of multiple small tasks (e.g., microservices) that are scheduled and deployed by a cloud orchestrator (e.g., Kubernetes [73]) across distributed nodes. Deployed tasks running on CPUs are isolated by guest sandboxes (VMs [4, 100], containers [34, 160]) and can be dynamically evicted, resumed, and migrated by the orchestrator for load balancing and auto-scaling. Each task is built as an OCI image [32] containing its executable and guest sandbox image.

FPGA architecture. In cloud infrastructures, FPGAs serve as standalone PCIe devices connected to each machine node [6, 28, 103]. The device contains FPGA fabric and onboard peripherals such as DDR memory, network ports, and a PCIe slot. A portion of the FPGA fabric is statically configured as a *Shell* [54, 157], which offers glue logic to connect to the onboard peripherals, such as a PCIe bridge, DMA, and network controller. The rest serves as a dynamic region for runtime custom logic reconfiguration.

FPGA applications. Like other accelerators, FPGAs are mainly used as coprocessors, where CPU applications offload their compute-intensive part and receive results after completion. Such FPGA applications consist of two parts: the offloaded computation part called *kernel code* and the main CPU application called *host code*.

The kernel code represents custom logic (or *user logic*) that executes the offloaded task. The kernel code can be written in any language supported by the underlying FPGAs (e.g., HDL [16, 136], HLS [10, 66, 83, 92, 119], and DSLs [11, 26, 84, 93]). The code is compiled using FPGA vendors' IDEs (e.g., Vivado [154]), which generate *bitstreams*. The bitstream is programmed to the FPGA's dynamic region to instantiate user logic there.

The host code is responsible for FPGA device management (e.g., reconfiguration) and task offloading via various vendor-provided APIs [55, 66, 158]. We primarily target the OpenCL API [66] because it is widely adopted by FPGA vendors [57, 157], and other APIs (e.g., XRT ([158]), OneAPI ([55])) inherit its programming model.

2.2 Design Challenges and Key Ideas

FPGA adoption in cloud-native ecosystems has not been fully explored by existing studies targeting traditional cloud instances [97, 161] or serverless platforms [38, 47]. We identify three key features to realize successful FPGA integration into cloud-native architectures, filling a gap in prior studies. First, *a lightweight FPGA virtualization mechanism* is essential, which abstracts physical FPGA devices from guest cloud-native applications (host code) running inside CPU sandboxes without compromising their lightweightness. This feature facilitates sharing limited FPGA resources among multi-tenant applications and also enables the orchestrator to transparently allocate/deallocate FPGA resources. Second, *an FPGA state*

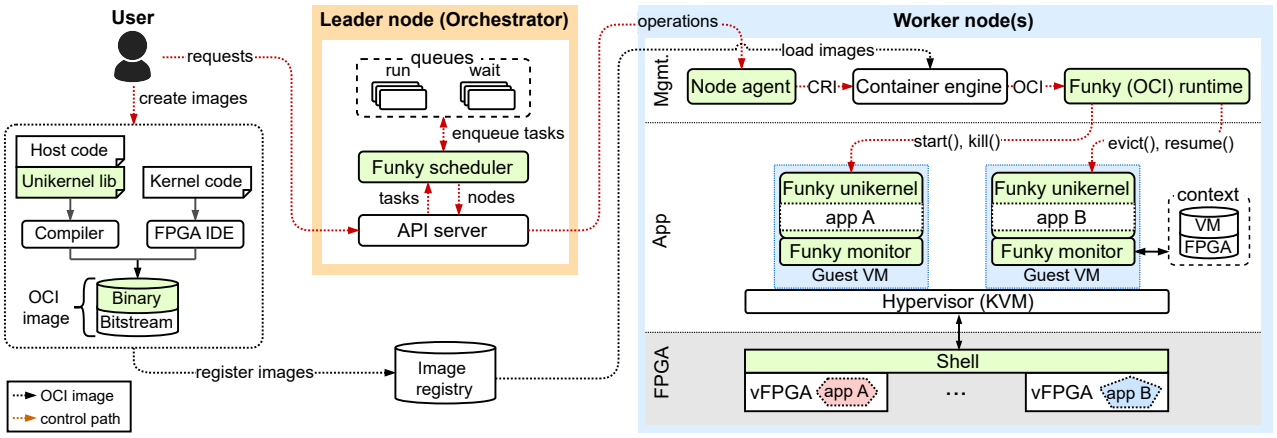


Figure 1: The Funky architecture for cloud-native FPGA orchestration. The key components are highlighted as green boxes.

management mechanism is critical for core orchestration operations such as preemptive scheduling, checkpointing, and load balancing. This feature must allow the orchestrator to safely save and restore application contexts residing in both CPU and FPGA devices. Lastly, we need *an end-to-end FPGA orchestration mechanism* that can be integrated into industry-standard orchestrators such as Kubernetes. This demonstrates the compatibility and applicability of our solutions within modern cloud-native ecosystems.

The FPGA-aware orchestration ecosystem must overcome design challenges related to FPGA constraints and prior solutions' limitations. We present four core challenges and Funky approach.

#1: Lightweight FPGA virtualization. Existing lightweight sandboxes [4, 87, 100, 160] only support virtualization for legacy devices such as disk and network, and FPGA support is missing. Prior studies propose FPGA virtualization for traditional cloud instances [97, 161], which are impractical for cloud-native environments due to a lack of reconfiguration support [97] or the complexity of the guest sandbox/API layer [161]. While rich sandboxes ensure system compatibility, their layered stacks obscure the interaction between guest applications and FPGAs, complicating orchestration services depending on their states, e.g., preemption.
Key idea. We design a new unikernel architecture virtualizing FPGAs for cloud-native applications, ensuring the lightweight context, near-zero overhead, and hypervisor-enforced isolation (see § 3.2).

#2: Application portability. Packaging FPGA applications as unikernel images imposes another challenge because of the lack of programmability. To address this issue, we aim to support a modern programming API for FPGAs, i.e., OpenCL, to retain application portability. Standard VMs or containers are capable of porting the entire FPGA software stacks to virtualized environments [97, 152]. However, porting the vendor OpenCL stack to the unikernel enlarges application contexts and undermines its lightweight architecture.
Key idea. We introduce FunkyCL, a lightweight OpenCL-compatible library that gains programmability for OpenCL execution workflows with minimal code changes (see § 3.3).

#3: FPGA task preemption and state management. Although cloud service preemption, migration, and checkpointing are active research areas [1, 36, 41, 79, 80, 120, 125, 138], they do not take accelerator device states (FPGAs, GPUs) into account. Prior studies propose hardware-assisted checkpoints for FPGA preemption [82, 105, 124], which integrate monitoring circuits into user logic to capture the FPGA's internal states. However, saving only the logic states is insufficient because the context of FPGA applications is distributed across CPUs, FPGAs, and their memories. This state distribution and FPGA's asynchronous execution model [66, 158] complicates orchestrators' ability to maintain the entire state of FPGA workloads.
Key idea. We propose an end-to-end state management mechanism for FPGA task preemption and checkpointing. The thin hypervisor transparently maintains both VM and FPGA contexts (see § 3.4).

#4: FPGA-aware orchestration. Integrating Funky's FPGA virtualization and preemption mechanisms into cloud-native orchestrators poses two design challenges. First, industry-standard orchestrators, e.g., Kubernetes, do not support checkpoint and restore operations even without FPGAs. Although the Kubernetes scheduler supports preemption and eviction for a group of containers (i.e., pods) [74], these functions terminate the containers and abandon their contexts. Second, the preemption requests must be propagated from the orchestrator to worker-node components while following the CRI/OCI specifications [31, 33] for system compatibility.
Key idea. We design CRI/OCI-compatible orchestration system components that leverage the proposed FPGA virtualization and state management mechanisms (see § 3.5).

3 Design

3.1 System Overview

Cloud-native architecture. Figure 1 illustrates the Funky's cloud-native ecosystem. We target an industry-standard cloud-native system stack following the CRI and OCI specifications [31, 33]. The cloud orchestrator runs on the leader node, whereas per-node system components and user applications run on the worker nodes. Every worker node is equipped with one or more PCIe-connected FPGAs, and each FPGA offers a dedicated Shell, onboard memory, and one or more reconfigurable slots (vFPGAs) where user logic is programmed. The orchestrator is responsible for deploying applications, managing the entire cluster, and enforcing scheduling policies. The cloud users push OCI images of applications to the

image registry before sending deployment requests to the orchestrator. The node agent runs on every worker node and forwards requests from the orchestrator to the container engine through CRI APIs. Upon API calls from the agent, the engine invokes the corresponding commands of the OCI runtime.

We highlight three system components extended/added to realize FPGA-aware orchestration. *Funky scheduler* is the extended scheduler component, which selects a worker node appropriate for deploying/migrating applications based on the applied policy. *Funky runtime* is an OCI-compatible runtime responsible for deploying, evicting, resuming, and migrating applications in response to orchestration requests. *Funky monitor* is a thin hypervisor layer that virtualizes FPGAs on worker nodes from unikernel applications.

Build process and tool chain. Figure 1 also illustrates a workflow of packaging applications as deployable images on the Funky architecture. Cloud users prepare an application binary and corresponding FPGA bitstreams. The binary is generated by compiling the host code statically linked with the Funky unikernel library, which exposes guest APIs for FPGA control to applications. The bitstream is generated from the kernel code with FPGA development tools such as Vivado [154]. The kernel code does not change because Funky runs upon the original vendor-provided FPGA platform [157].

3.2 Lightweight FPGA Virtualization

We design the Funky unikernel, a lightweight, isolated guest sandbox virtualizing distributed FPGAs. The Funky unikernel virtualizes distributed FPGAs in an orchestrator-friendly manner. First, the lightweight state of a unikernel mitigates start-up latencies and virtualization overheads. Second, its simplified state and single-process basis facilitate FPGA task preemption and migration across distributed nodes. The underlying hypervisor can easily trace memory transfers and operations among the guest application and FPGAs.

The Funky unikernel delegates most FPGA-related jobs requested by applications to a host-side hypervisor layer. This approach is similar to API remoting [150, 161]. However, the Funky unikernel does not directly delegate APIs but converts them to minimal I/O requests essential for FPGA task offloading (e.g., data copy and synchronization) to securely isolate guest applications and the host system stack.

Funky's FPGA virtualization mechanism is technically applicable to other hypervisor-based sandboxes, e.g., standard full-blown VMs and Firecracker's microVMs [4]. Funky adopts a unikernel as a guest sandbox because of increasing security concerns in cloud-native environments, given that even containers are running inside VMs (e.g., Kata container [108]). We strive to design the unikernel-based virtualization mechanism without losing the security properties of hypervisor-enforced isolation.

Unikernel design. Figure 2 shows the Funky unikernel architecture. *The Funky unikernel* virtualizes reconfigurable FPGA slots as *vFPGAs* and allows applications to request managing vFPGAs through guest APIs. The unikernel exposes *vFPGA layer* to a guest application and library, virtualizing the host FPGA and offering abstracted interfaces: hypercalls and exitless I/O [44, 134]. The guest FPGA library converts API calls from the guest application to either the hypercalls or exitless I/O requests. In this work, we design *FunkyCL*, a lightweight OpenCL-compatible guest library (§ 3.3).

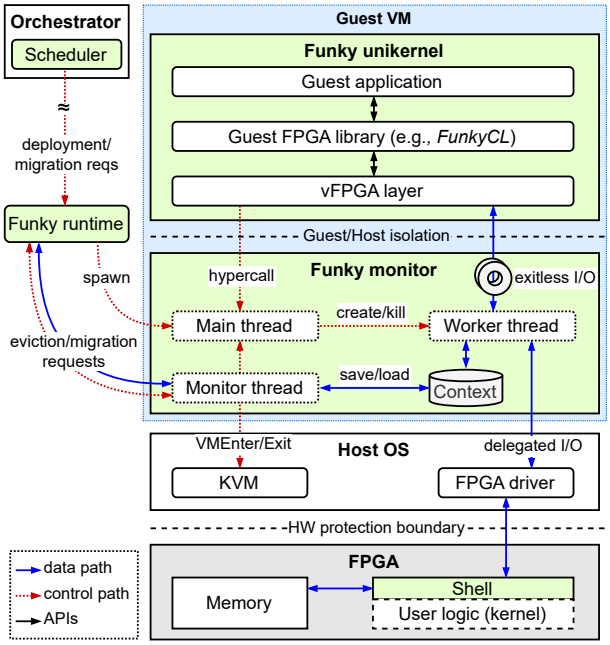


Figure 2: Funky's unikernel architecture.

The Funky monitor is a host-side user process serving as a thin hypervisor layer for each Funky unikernel, similar to QEMU [14]. The Funky monitor is responsible for launching the corresponding unikernel and monitoring its states. It is also responsible for handling requests from the guest application and the orchestrator while ensuring host-side isolation among multiple Funky monitors. To handle these asynchronous requests efficiently, the monitor creates two threads: a *worker thread* and *monitor thread*. The former handles FPGA control requests from the guest and manages the underlying FPGA via a host FPGA system stack (e.g., Xilinx Runtime [158]). The latter handles migration/eviction requests from the orchestrator and triggers saving/loading guest VM and FPGA contexts.

Hypercalls. The Funky unikernel offers only two hypercalls used for vFPGA allocation. vfpga_init() requests the Funky monitor to search for an available vFPGA and assigns it to the guest application. The hypercall also transfers a bitstream to let the Funky monitor reconfigure the vFPGA. Subsequently, the Funky monitor creates the worker thread to asynchronously handle requests from the guest. The requests are forwarded via lock-free message queues shared between the unikernel and the monitor. The Funky unikernel initializes the queues in the guest memory space and notifies their addresses to the monitor via the hypercall. Conversely, vfpga_exit() releases the obtained vFPGA and deletes the worker thread.

Asynchronous I/Os. The Funky unikernel offers a fast, exitless I/O interface for FPGAs to avoid frequent context switches. To realize this, we design Funky requests, FPGA-related primitive operations to be handled on the underlying FPGA platform. Table 2 shows four primary Funky requests. A guest unikernel sends the Funky requests to the worker thread via shared queues without invoking VMEXIT. The worker thread securely validates the received Funky requests and performs the delegated I/Os upon the requests. We note that the Funky requests listed in Table 2 are designed

OpenCL APIs	Standard OpenCL definition	FunkyCL definition
clCreateProgramWithBinary()	Creates a program object and loads bitstreams.	Calls vfpga_init() to configure user logic.
clReleaseProgram()	Decrements the program reference count.	Calls vfpga_free() if the count gets zero.
clCreateBuffer()	Creates an OpenCL buffer (memory) object.	Sends a MEMORY() request.
clEnqueueKernel()	Enqueues a command to execute a kernel.	Sends an EXECUTE() request.
clEnqueueMigrateMemObjects()	Enqueues a command to migrate memory objects.	Sends a TRANSFER() request.
clFinish()	Waits until all enqueued commands are complete.	Sends a SYNC() request.

Table 1: Definitions and functional changes of important OpenCL host APIs [66]; other APIs are omitted for space.

Request type	Description
MEMORY(*buff_id, src, size*)	Allocates a buffer on FPGA.
TRANSFER(*queue, buff_id, src, size*)	Invokes a data transfer between Host & FPGA memories.
EXECUTE(*queue, kernel, args*)	Invokes a kernel execution.
SYNC(*queue, req_id*)	Awaits request completion.

Table 2: Funky requests for FPGA operations.

for OpenCL support (§ 3.3). Other Funky requests can be further designed to support different programming models.

3.3 FunkyCL Library

We design FunkyCL, an OpenCL-compatible library that brings application portability and compatibility for Funky unikernel applications. FunkyCL is designed to realize two key functionalities. First, it is fully compatible with the native OpenCL specification standardized by Khronos group [66]. Second, FunkyCL APIs achieve performance equivalent to the native execution of OpenCL APIs.

OpenCL APIs. The FunkyCL library is part of the Funky unikernel library and allows the guest application to control the assigned vFPGA via the standard OpenCL APIs. The API calls issue the Funky requests described in Table 2. FunkyCL offers a library-level FPGA abstraction to enable transparent use of the underlying FPGAs; it exposes the Funky unikernel's vFPGA layer as an OpenCL device named *vFPGA*. OpenCL host APIs issued to the device are transparently converted to either hypercalls or Funky requests.

Table 1 lists important OpenCL APIs and functional changes in FunkyCL. clCreateProgramWithBinary() and clReleaseProgram() play an essential role: acquiring and releasing vFPGAs. Because creating a program object triggers FPGA reconfiguration, FunkyCL lets clCreateProgramWithBinary() invoke vfpga_init() to obtain the vFPGA. The Funky monitor handles the hypercall and spawns the worker thread if it finds an available vFPGA. Subsequently, the worker thread reconfigures the slot with a bitstream. The other OpenCL APIs in Table 1 send corresponding requests to the worker thread until the vFPGA is released by clReleaseProgram().

Zero-copy host-device data transfers. As a guest application runs in an isolated context in Funky, the host-device data transfer is challenging; such data transfers typically induce redundant data copies, such as shadow paging [2], to keep data consistent. FunkyCL avoids the memory management overhead thanks to the unikernel's single address space. Specifically, when a guest application calls any OpenCL API for data transfers, FunkyCL creates a TRANSFER() request, which only contains the address and data size of the guest memory buffer. The worker thread receives the request and then translates the guest address to the host only once. Finally, it transfers the buffer data to the FPGA memory or vice versa.

3.4 FPGA State Management

We design an FPGA state management mechanism, enabling task preemption/migration and checkpointing to improve scheduling fairness and fault tolerance. To maintain application contexts distributed across CPUs and FPGAs, Funky adopts a hypervisor-driven approach, i.e., driven by the Funky monitor. Since the Funky monitor is spawned per guest VM and handles delegated FPGA requests, it can easily trace both guest VM (CPU) and FPGA states.

Funky supports three state management commands: *checkpoint* that saves snapshots of the entire VM/FPGA contexts, *evict* that only evicts the FPGA context in host memory, and *resume* that resumes the task from either the saved snapshot or evicted context. These commands are triggered by an extended container runtime, i.e., Funky runtime (§3.5), in response to high-level orchestration operations requested by the orchestrator. To enable this, the Funky monitor spawns a *monitor thread* that exposes an inter-process communication (IPC) interface to let the Funky monitor issue these commands. The monitor thread is also responsible for saving and restoring the VM context and cooperating with the worker thread to maintain an FPGA context.

FPGA architecture. We first describe our target FPGA architecture and guest application states, illustrated in Figure 3. Funky primarily targets vendor-provided Shells [7, 57, 75]. The FPGA device consists of three hardware components: Shell, vFPGA(s), and off-chip FPGA memory. The Shell offers the PCIe bridge IP that exposes memory-mapped I/Os to host CPUs and the DMA controller for data transfers between the host and FPGA memory. The vFPGAs configure user logic, which could have control and status registers (CSRs) for the execution invocation. The FPGA memory (HBM, DRAM) and its controller are used as the main working memory, where we store input/output data consumed/produced by the kernel. In the architecture, the states of FPGA applications can be classified into three types: the FPGA logic (kernel), FPGA memory, and the guest VM.

FPGA synchronization. We next describe how these application states are saved by Funky. Because FPGAs do not support context switch mechanisms due to their complex and modular architectures, we cannot suspend any operations running on the FPGA, e.g., DMA transfers and kernel executions. Therefore, before saving the FPGA states, Funky tolerates waiting for all on-the-fly FPGA operations (i.e., Funky requests) to complete. Specifically, when the monitor thread receives checkpointing or eviction requests, it asks the worker thread to invoke the SYNC() request and makes the FPGA state (both user logic and Shell) *consistent*. We note that such synchronization operations do not increase the total execution time of running applications because any computation requested by the application is still running during the synchronization.

While the synchronization operations do not incur performance overheads, they may delay invoking eviction and migration requests and affect the latency of orchestration operations, particularly for a single, long-running request, e.g., processing a 1 GiB chunk. Therefore, Funky supports a mechanism to mitigate the synchronization time by splitting such a large request into multiple requests for smaller chunks, particularly for streaming computation that repeats the same operation for all inputs (e.g., FFT).

FPGA logic state. Following synchronization, the worker thread saves the FPGA's kernel and memory states. The logic state comprises CPU-readable control registers and temporal contents stored in on-chip elements (flip-flops, BRAM). While the latter is not accessible from the host, these internal states are ignorable for most of the state-of-the-art FPGA accelerators following the OpenCL execution model [50, 96, 106, 128, 142, 144, 163] because their internal states are flushed when the execution completes. Funky can technically support saving/restoring FPGA internal states by leveraging hardware-assisted FPGA checkpointing [9, 91, 105, 124].

FPGA memory state. Funky efficiently saves and restores FPGA memory contexts depending on the states of each memory buffer, via DMA data transfers. During the execution, the worker thread traces MEMORY() requests and reserves locations of memory buffers. Then, upon TRANSFER() request, it updates their states as follows:

- *init* – the buffer has no data in the FPGA memory.
- *sync* – the buffer is synchronized with a host buffer.
- *dirty* – the buffer is not synchronized with a host buffer.

When saving FPGA contexts, Funky only records the states of *dirty* memory buffers. Memory buffers whose state is either *init* or *sync* are ignored to mitigate the total context size. The monitor thread kills the worker thread after all the FPGA states are saved.

VM state. After saving FPGA contexts, the monitor thread optionally saves VM contexts. Funky adopts a simple hypervisor-driven approach for saving and restoring VM states. When saving the state as a snapshot, the monitor thread first triggers an interrupt to the vCPU, causing VMEXIT. It then captures the context of the vCPU (e.g., CPU registers) and dirty pages in guest memory. When restoring the state, the thread initializes the vCPU and guest memory contexts with the selected snapshot and lets the vCPU invoke VMENTER. The eviction process does not save the VM state to the disk but keeps it in the host memory to mitigate the overhead.

Restoring states. There are two ways to restore the application state from saved contexts and resume the execution. First, if only the FPGA states are evicted, the monitor thread simply respawns the worker thread with the saved context and lets it restore the register and memory states of FPGA kernels via memory-mapped I/O and DMA transfers. The worker thread then notifies the completion of a SYNC() request to the guest application to resume the execution. Second, if the application is resumed from the entire snapshot, the Funky monitor is spawned with the snapshot and first restores the VM state in CPU memory, which includes some of the FPGA contexts, e.g., contents of *sync* buffers. Then, it spawns the worker thread and restores FPGA states as explained.

Memory management and isolation. The Funky monitor guarantees FPGA memory isolation across guest VMs for multi-tenancy. The Funky monitor restricts access to vFPGAs and FPGA memory buffers that are allocated to other applications. To mitigate side-channel risks, the Funky monitor zeros the memory buffers when a guest application finishes.

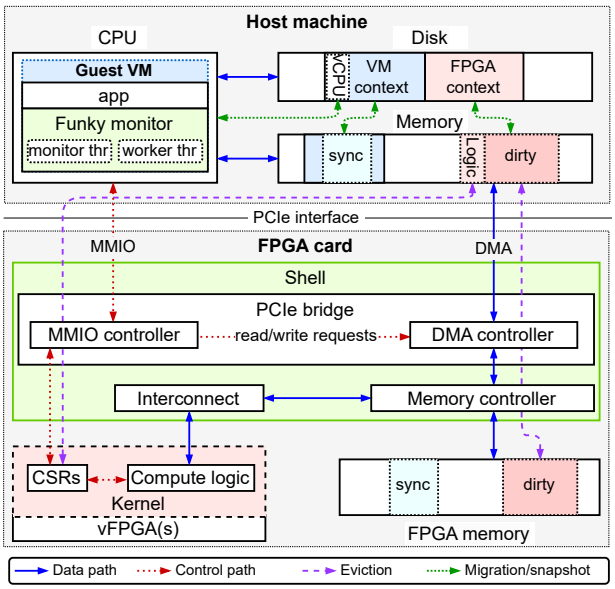


Figure 3: Funky's hardware architecture.

3.5 FPGA-aware Orchestration

Lastly, we design an FPGA-aware orchestrator that leverages Funky's FPGA virtualization and state management mechanisms. Assuming that the use of FPGAs is selective depending on workloads, we strive to achieve two design goals. First, our design can adapt existing industry-standard orchestrators (e.g., Kubernetes) by reasonably extending them without violating CRI and OCI specifications. Second, the extended orchestrator can distinguish between FPGA and CPU workloads, applying Funky-specific features only to FPGA ones.

Orchestration Components. To integrate Funky features into industry-standard orchestrators, we extend three system components: the scheduler, node agent, and OCI runtime.

The Funky scheduler is an orchestration component for FPGA task deployment and preemption. It adopts the same task eviction mechanism as the Kubernetes scheduler [74], evicting running tasks if they occupy resources (e.g., vFPGAs) requested by high-priority tasks. However, unlike the Kubernetes scheduler that always terminates evicted tasks, Funky keeps the context of evicted tasks and transparently deallocates resources from them. The evicted tasks can be either *resumed* on the same node or *migrated* to another node.

Node agents propagate requests from the orchestrator to the container engine through CRI APIs. To forward Funky-specific information, we leverage *annotations*, unstructured key-value pairs defined in the CRI API's message structure. The node agents attach FPGA-specific metadata to annotations of primary CRI APIs, allowing the container engine to invoke corresponding Funky runtime commands without violating the CRI specification.

Funky runtime is an OCI-compliant low-level container runtime that maintains a lifecycle of Funky unikernels. In addition to commands defined in the OCI specification [33], it supports five Funky-specific commands: evict, resume, checkpoint, replicate, and update.

Orchestration Services	Operations	CRI API (*key metadata*)	Funky OCI runtime command
Preemptive scheduling	Deploy	CreateContainer (*preemptible**) → StartContainer (*cid*)	create <*cid*> → start <*cid*>
	Evict	StopContainer (*cid*)	evict <*cid*>
	Resume	StartContainer (*cid*)	resume <*cid*>
	Migrate	CreateContainer (*cid*, node_id**) → StartContainer (*cid*)	resume <*cid, node_id*>
Checkpoint & restore	Checkpoint	CheckpointContainer (*cid*)	checkpoint <*cid*>
	Restore	CreateContainer (*cid*, node_id**) → StartContainer (*cid*)	resume <*cid, node_id*>
Workload scaling	Horizontal	CreateContainer (*cid*, node_id**) → StartContainer (*cid*)	replicate <*cid, node_id*>
	Vertical	UpdateContainerResources (*cid, vfpga_num**)	update <*cid, vfpga_num*>

Table 3: Funky's orchestration services. *cid* abbreviates *container id*. * represents metadata attached by *annotations*.

Algorithm 1: Funky's preemptive task scheduling.

```
1  schedule(wait_queue, run_queue, nodes)
2  begin
       /* Pick the most prioritized task */
3      task ← pull(wait_queue);
       /* Find the most suitable node for the task */
4      node ← schedule(task, nodes);
       /* Evict, resume, migrate, or deploy the task */
5      if node is occupied by other tasks then
6          evicted_task ← evict_req(task, node, run_queue);
7          push(evicted_task, wait_queue);
8          deploy_req(task, node);
9      else if task is evicted and task is on node then
10         resume_req(task, node);
11     else if task is evicted and task is not on node then
12         migrate_req(task, node);
13     else
14         deploy_req(task, node);
15     push(task, run_queue);
16 end
```

Orchestration services. Table 3 summarizes Funky's primary orchestration services and associated operations. To distinguish FPGA and non-FPGA tasks, Funky attaches a *preemptible* flag to deployed FPGA tasks. Any orchestration requests for *preemptible* tasks are propagated to the Funky runtime to invoke Funky operations. The *node_id* represents a worker node where the context of the target task exists. When migrating, restoring, or replicating a task, the Funky runtime communicates with the other runtime on a remote node specified by *node_id* to obtain the task context. *vfpga_num* represents the maximum number of vFPGAs allocatable to a task, which is used for vertical scaling.

Preemptive task scheduling. Algorithm 1 details the workflow of Funky's preemptive task scheduling. The Funky scheduler has a *run queue* holding running tasks and a *wait queue* holding submitted/evicted tasks sorted by their priorities. The scheduler picks up a task to schedule next from the wait queue and selects the most suitable node to place the task (L3-4). The scheduler then *evicts*, *resumes*, *migrates*, or *deploys* the task depending on its state. When other running tasks occupy the selected node (L5-8), the scheduler *evicts* any of them; it moves the task from the run queue to the wait queue and sends the eviction request to the node agent. The evicted task state is reserved on the same worker node until resumed or migrated. When rescheduling an evicted task, depending on the selected node, the scheduler either *resumes* the task on the same node (L9-10) or *migrates* the task to the remote node (L11-12).

Checkpoint and restore. Funky supports both automatic and manual checkpointing functions. In the former case, the orchestrator periodically takes snapshots of the target task, and if the orchestrator detects task failures, it attempts to restore it from the latest snapshot. In the latter case, these operations are manually invoked upon user requests on demand. The failed tasks can be restored on either the current (local) node or any remote node. The execution flow is the same as the *evict* and *migrate* of the preemptive scheduling.

Workload scaling. Funky supports both horizontal and vertical scaling for FPGA workloads. For the horizontal scaling, Funky *replicates* a running task by saving and copying its context. However, if the target task is still alive and running, the container engine invokes *replicate* operation to deploy the replicated task on the node represented by *node_id*. For the vertical scaling, Funky increases the task's limit of allocatable vFPGAs given by *vfpga_num*.

4 Implementation

We build a prototype of the Funky framework targeting the AMD Vitis platform [157] built upon the Alveo U50 FPGA [151] with the XDMA Shell, which serves a single reconfigurable slot.

Funky unikernel and FunkyCL. We implement the Funky unikernel and FunkyCL library based on the IncludeOS unikernel [17]. We port open-sourced OpenCL headers [65] and C++ bindings [64] to IncludeOS and implement their entities as a part of the static unikernel library. Some FunkyCL APIs do not issue requests to the backend to mitigate the communication overhead. For example, kernel arguments set by clSetKernelArg() are transferred together through the EXECUTE() request.

Funky monitor. We implement the Funky monitor based on the Solo5 [147] hypervisor, which consists of frontend bindings that offer application binary interfaces (ABI) to the guest unikernel, and a backend hypervisor layer (called ukvm) that acts as a vCPU thread. We mainly extend ukvm to implement the core functionalities, including the FPGA worker thread. The FPGA worker thread handles Funky requests from the guest through APIs offered by Xilinx Runtime (XRT) [158]. We also implement two hypercalls, vfpga_init() and vfpga_free(), for obtaining and releasing FPGA slots.

Orchestrator and runtime. Our orchestrator prototype consists of the API server handling orchestration service requests (e.g., Deploy, Checkpoint) and the scheduler for preemptive scheduling. It propagates the respective operations to the Funky runtime daemons on worker nodes. The Funky runtime then issues corresponding commands shown in Table 3.

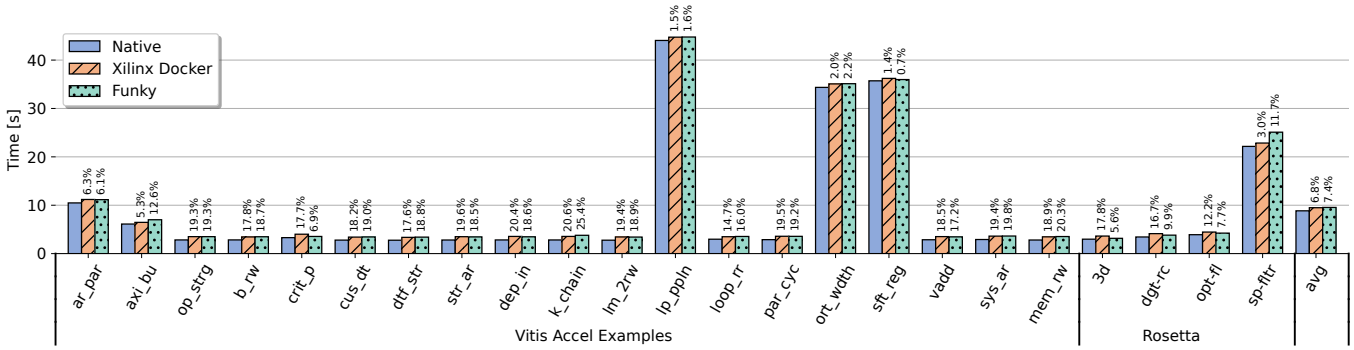


Figure 4: End-to-end application execution time for native execution, Xilinx Docker container [155], and Funky, with relative gaps (%) to the native execution.

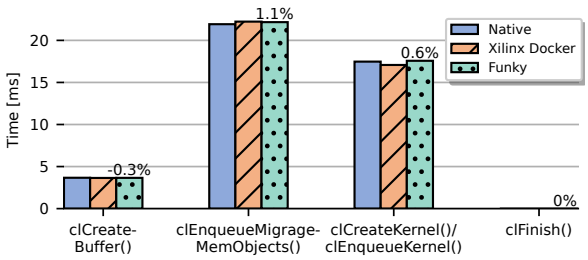


Figure 5: OpenCL API times with relative gaps (%) to the native execution.

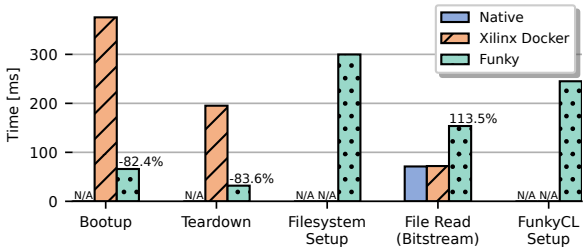


Figure 6: Funky's setup overheads with relative gaps (%) to the Xilinx Docker container.

5 Evaluation

We comprehensively evaluate Funky's virtualization overheads (§ 5.2), portability (§ 5.3), checkpointing (§ 5.4), task preemption (§ 5.5), and end-to-end orchestration (§ 5.6).

5.1 Experimental Setup

Testbed. We set up four x86 servers, where three servers as worker nodes are powered by an Intel Xeon Gold 6238R Processor running at 2.2 GHz and one as a leader node powered by an Intel Xeon Gold 5317 Processor running at 3 GHz. All servers connect through a 100 Gbps switch. The worker nodes run Ubuntu 20.04, which are equipped with 256 GiB DDR4 at 2933 MHz, a 960 GiB SATA SSD as persistent storage, and an Alveo U50 FPGA via a PCIe Gen3 x16 bus.

Benchmark suites. Since Funky currently supports the OpenCL programming model, we naturally use benchmark suites written in OpenCL. We choose two benchmark suites: Vitis Accel Examples [153] and Rosetta [164]. Vitis Accel Examples is one of the most suitable benchmark suites as it is officially maintained by AMD and offers various applications to test a broad range of OpenCL APIs. We particularly use cpp_kernels/ of Vitis Accel Examples, containing a set of primary computation kernels, such as vector add, matrix multiplication, and an FIR filter. Rosetta is a widely used benchmark suite designed for AMD Xilinx FPGAs, offering practical, real-world FPGA workloads, including 3D rendering, digit recognition, optical flow, and spam filtering.

Baselines. We compare Funky with two baseline setups: native execution and an industry-standard Docker container. For the container setup, we use Xilinx Base Runtime [155], an FPGA-accessible container officially maintained by AMD. The container runs Ubuntu 20.04 and installs a full XRT package, which directly communicates with the host-side FPGA device drivers.

5.2 Virtualization Overheads

Firstly, we clarify Funky's FPGA virtualization overheads.

Methodology. We measure each application's execution time in Funky against the two baselines. In our experimental setup, we allocate 1 GiB of memory for each unikernel. We also break down Funky's virtualization overheads to understand their source. Specifically, we analyze the runtime overheads of OpenCL APIs, which are core FPGA operations, and initial setup overheads to create and destroy execution sandboxes, i.e., unikernels and containers. We use wide_mem_rw for the microbenchmarking.

Results. Figure 4 shows the end-to-end execution time of all the applications. Figures 5 and 6 show the breakdown of OpenCL API and initial setup overheads. As shown in Figure 4, Funky incurs only a 7.4% overhead on average compared to native execution, which is close to the container setups (6.8%). We highlight that Funky does not incur additional overheads for FPGA operations through OpenCL API calls as shown in Figure 5. The performance gaps from the baselines come from the initial setup overheads of execution sandboxes. In Figure 6, for the Xilinx containers, their bootup/teardown latencies are the main factor of the performance penalties, while Funky unikernels cut these overheads by 82.4% and 83.6%. Although we observe other setup overheads specific to Funky, the main factor is not related to FPGA operations but heavy file I/O operations of IncludeOS's filesystem, causing the biggest slowdown for spam-filter (sp-fltr). Only the FunkyCL setup time, 245.1 ms on average, stems from Funky's extension, where

	Application	Codebase			OCI image [MiB]		
		LoC	diff	(fop)	Funky	Cont	bs
Vitis_Accel_Examples	array_partition	150	2	(1)	32.7	1131.8	29.6
	axi_burst_performance	126	8	(7)	66.2	1165.5	62.9
	bind_op_storage	77	2	(1)	32.7	1131.9	29.7
	burst_rw	73	2	(1)	32.6	1131.7	29.5
	critical_path	92	4	(3)	36.9	1136.2	29.9
	custom_datatype	193	4	(3)	32.4	1131.7	29.1
	dataflow_stream	75	2	(1)	32.3	1131.4	29.3
	dataflow_stream_array	81	2	(1)	32.4	1131.6	29.4
	dependence_inter	99	2	(1)	32.5	1131.6	29.4
	kernel_chain	195	6	(5)	33.4	1132.6	30.3
	lmem_2rw	76	2	(1)	32.6	1131.7	29.6
	loop_pipeline	119	2	(1)	32.8	1131.9	29.7
	loop_reorder	96	2	(1)	33.9	1133.0	30.8
	partition_cyclicblock	152	2	(1)	33.6	1132.7	30.5
	port_width_widening	175	2	(1)	34.5	1133.6	31.4
	shift_register	152	5	(4)	32.9	1132.1	29.9
	simple_vadd	109	18	(17)	32.5	1131.4	29.5
	systolic_array	102	2	(1)	35.1	1134.2	32.0
	wide_mem_rw	77	2	(1)	33.2	1132.2	30.0
	common lib	754	95	(53)	-	-	-
Rosetta	3d-rendering	3456	1	(0)	34.4	1132.3	29.6
	digit-recognition	217	13	(12)	36.1	1134.8	32.5
	optical-flow	1624	75	(74)	60.4	1146.6	31.4
	spam-filter	387	26	(25)	114.0	1213.8	30.7
	common lib	475	31	(31)	-	-	-
	Average	-	3.4%	(2.7%)	39.6	1138.2	-

Table 4: Portability study. diff is the total lines changed from the original code, and (fop) is only for file operations. common lib is code sets shared among each benchmark suite. Funky and Cont are Funky unikernel and Xilinx Docker container image sizes. bs is the total size of bitstreams.

Funky copies the bitstream data for eviction/migration requests and spawns the FPGA worker thread.

In summary, Funky induces a modest one-time setup overhead for FPGA virtualization and state management.

5.3 Application Portability

Secondly, we study Funky's application portability, i.e., how much effort developers require to port their FPGA applications.

Methodology. We compare the Line of Code (LoC) for each application before and after porting to Funky. We also analyze the OCI image sizes of Funky unikernels and Xilinx Docker containers.

Results. Table 4 shows the summary. The total code change is just 3.4% on average. We note that developers need to add just one line to the native code, #include <os>, to use the OpenCL API. Most changes relate to modifying file API calls for IncludeOS, 2.7% in total. Although IncludeOS advertises traditional file API support (e.g., fopen), we encounter various portability issues and replace file operations with the IncludeOS-specific file API (memdisk). Other changes are due to omitting vendor-specific OpenCL extensions [156].

Regarding OCI image sizes, Funky unikernel is drastically smaller (28.7× on average) than the container. Funky's OCI image only contains the application binary, bitstream, input datasets, and minimal unikernel components to execute the application (e.g., FunkyCL). The bitstream and datasets are dominant in Funky's image, and the unikernel's execution binary itself is only 3-4 MiB. In contrast, the Xilinx Docker container significantly increases its image size due to the XRT package and its dependent libraries.

In summary, Funky enables porting OpenCL applications without changing FPGA-related code. Moreover, Funky's OCI image is substantially smaller than a vendor-provided container image, which eases to maintain their instances across distributed nodes.

5.4 State Management

Thirdly, we evaluate Funky's state management mechanisms.

Methodology. In this experiment, we use shift_register as a microbenchmark and measure the latency of each state management operation: FPGA eviction, VM migration, and checkpointing, while changing its input data size. Note that resuming/restoring tasks triggers FPGA reconfiguration, which takes around 3.5 seconds. We omit this overhead because it comes from a hardware limitation of the underlying Shell (Vitis XDMA), e.g., Coyote [85] can shorten it to 20 ms. Such hardware optimization is out of our paper's scope.

FPGA state eviction. We first measure the time to evict and resume FPGA states. We trigger the eviction requests just after the kernel invocation is finished, ensuring that input and output (dirty) buffers exist on the FPGA memory. We change the input dataset size from 1 MB to 1000 MB.

Figure 7 shows the results, highlighting that FPGA eviction and resumption take 177.2 and 340.8 ms for the biggest data size (1000 MiB for each input/output). The eviction is significantly faster for small datasets, i.e., 0.4 ms for 1 MB. For the eviction, Funky only saves dirty buffers' states, leading to less overhead than evicting all buffers. FPGA resumption takes longer than the eviction due to (1) a consistent overhead of the worker thread creation (97.6-158.0 ms) and (2) transferring both input and output buffers.

In summary, Funky can reasonably evict and resume FPGA states in the order of milliseconds, even for bigger datasets (1000 MiB).

VM migration and checkpointing. Next, we evaluate the time to save and load the entire VM state. We measure the VM saving and restoring overheads in two cases: migration (VM save, VM load), saving snapshots into host memory as a temporal cache, and checkpointing (Checkpoint, Restore), saving them into persistent storage (SSD).

Figure 8 shows the overhead breakdown. As input/output data sizes increase, the snapshot sizes also increase (125.5 MiB to 2.1 GiB). We observe that the overhead for each operation reasonably increases along with increasing input data size. Checkpoint is slower than Restore because it iterates over the whole guest memory to find dirty pages and individually writes them to the disk, leading to slower random accesses. Notably, FPGA-specific operations do not increase the overall time of VM migration and checkpointing; the FPGA eviction occupies 0.4-10.6% in VM save and 0.1-2.4% in Checkpoint. While the FPGA resumption gets dominant for small data sizes due to its static overheads, it gets amortized for large applications.

In summary, Funky's checkpoint mechanism induces reasonable overheads to make a snapshot of FPGA workloads, which enables fast recovery from system failures.

FPGA synchronization. Lastly, we evaluate the synchronization overheads and our data splitting optimization to mitigate them

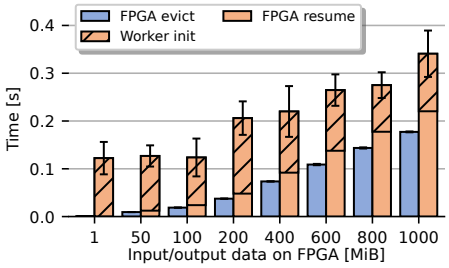


Figure 7: FPGA evict/resume overheads.

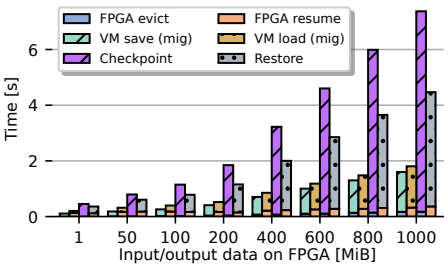


Figure 8: VM migration/checkpointing.

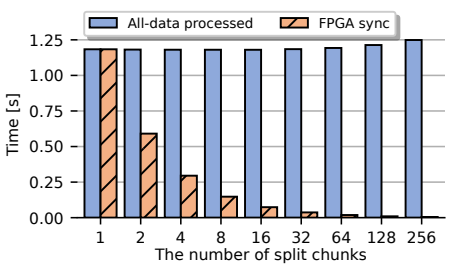


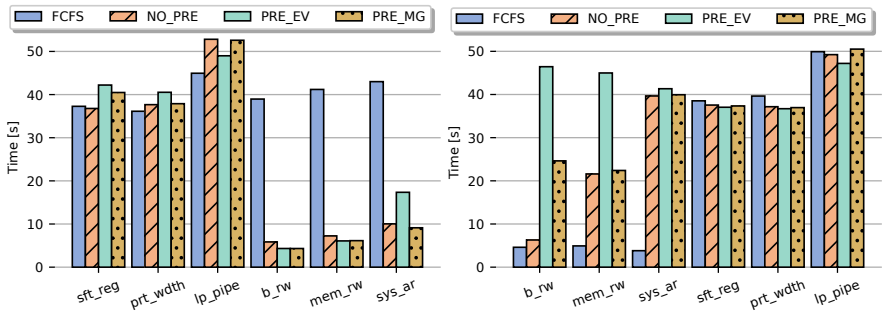
Figure 9: FPGA synchronization time.

Policy	Description	Evict	Migrate
NO_PRE	Reorder enqueued tasks.		
PRE_EV	Evict low-priority tasks.	✓	
PRE_MG	Migrate evicted tasks.	✓	✓

Table 5: Scheduling policies.

App	Time	Type	Priority per scenario	
			Short-HP	Long-HP
shift_reg	36.0 s	Long	Low	High
port_width	35.1 s	Long	Low	High
loop_pipe	44.8 s	Long	Low	High
burst_rw	3.5 s	Short	High	Low
mem_rw	3.5 s	Short	High	Low
sys_array	3.6 s	Short	High	Low

Table 6: Execution time and priority in each scenario.



(a) Short-HP scenario. **(b) Long-HP scenario.**

Figure 10: Task preemption effectiveness. FCFS shows the worst case when tasks are deployed in the x-axis order, which lets low-priority tasks occupy FPGAs.

(§ 3.4). This experiment fixes the total input data at 1000 MiB and splits it into 1 to 256 chunks, measuring the overhead as the kernel is repeatedly invoked/synchronized until all the chunks are processed.

Figure 9 shows the total execution time (All-data processed) and FPGA synchronization time (FPGA sync) in the worst-case scenario, where the synchronization request is invoked immediately after the kernel invocation. The results highlight that our optimization can cut 96.9% of the overhead without any performance overheads (<0.1%) in the case of 32 chunks, where each kernel invocation processes 31.25 MiB data. We also observe that excessive data splitting increases the total time, e.g., 5.5% with 256 chunks. Funky can prevent it by configuring the lower boundary of the chunk size.

In summary, the results demonstrate that our optimization can effectively eliminate the waiting time for FPGA synchronization.

5.5 Task Preemption

Next, we examine how Funky's state management mechanisms are effective for preemptive task scheduling on our FPGA cluster.

Methodology. We execute our orchestrator's prototype on the leader node and the Funky runtime daemons in the other three worker nodes. Table 5 shows the priority-based scheduling policies we compare. Every policy deploys tasks in the scheduling queue in a First-Come-First-Served (FCFS) manner as long as any FPGA is free. If all the FPGAs are occupied by running tasks, NO_PRE sorts the waiting queue based on the task's priority. On the other hand, PRE_EV evicts a low-priority task when a higher-priority task comes. PRE_MG performs both the eviction and migration; the migration takes place when an evicted task can be resumed on a different node.

We create two scenarios where we assign priorities to each task considering its execution time: **1. Short-HP** prioritizes short-lived tasks labeled as high priority (HP), and **2. Long-HP** prioritizes long-running tasks. We use a subset of Vitis Accel Examples, which contains six applications with long and short execution times, as shown in Table 6. We perform the batch execution in both scenarios, repeatedly sending deployment requests of the six applications while changing their orders to cover all possible deployment orders of high- and low-priority tasks. We test 20 patterns of different orders and measure the average execution time for each benchmark.

Results. Figure 10 (a) shows the behavior of the different policies in the Short-HP scenario, where PRE_MG is most effective for high-priority tasks, reducing 16.7% of their average execution time compared to NO_PRE. In the case of non-preemptive policies (FCFS and NO_PRE), we observe that high-priority tasks (b_rw, mem_rw, sys_ar) take longer execution time because they have to wait for the completion of low-priority tasks when all three FPGAs are in use. On the other hand, when preemption is enabled (PRE_EV and PRE_MG), the long-running tasks get evicted in favor of high-priority tasks. There is one edge case, sys_ar for PRE_EV, whose deployment is impeded by long FPGA synchronization of lp_pipe consisting of only two heavy FPGA operations. Such a case can be mitigated by our optimization demonstrated in Figure 9.

Figure 10 (b) shows the Long-HP scenario, where PRE_EV is most effective for high-priority tasks: 2.2% faster than NO_PRE. PRE_MG is less effective because evicted low-priority tasks are sometimes migrated before high-priority tasks arrive, slightly delaying their deployment. This is a common issue of task preemption, which can be mitigated by adapting better algorithms [25, 40, 60].

In summary, the evaluation demonstrates that Funky enables FPGA task preemption across distributed FPGA nodes, leading to shorter execution times than non-preemptive scheduling.

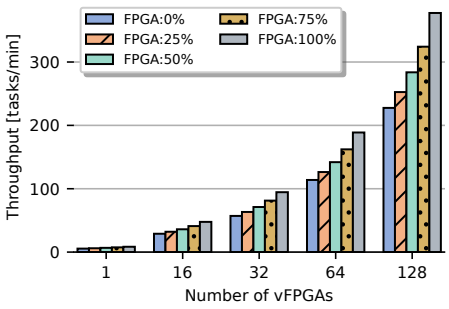


Figure 11: Workload scaling.

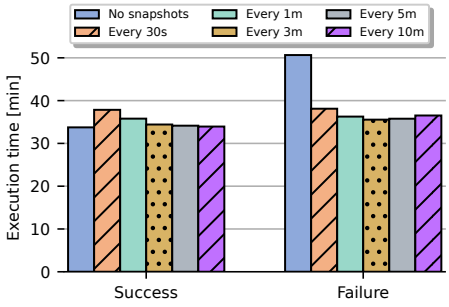


Figure 12: Fault tolerance analysis.

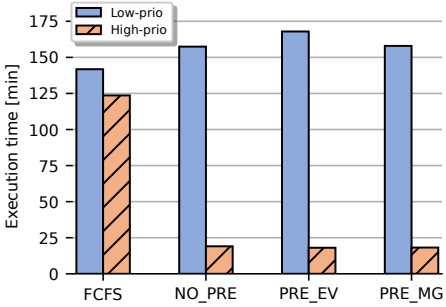


Figure 13: Preemptive task scheduling.

5.6 End-to-end Orchestration

Lastly, we evaluate the effectiveness of Funky's orchestration services. Due to our limited number of FPGA resources, we simulate a large-scale FPGA cluster using production traces from Google.

Methodology. We use production traces from the Google ClusterData 2019 dataset [63]. The dataset comprises one month of real-world application traces (execution, failure, and eviction) from eight Google Borg clusters [137, 140].

Because the traces only refer to CPU workloads, we strive for trace modification to reasonably simulate the behavior of FPGA-accelerated jobs. First, we manipulate the execution time of each trace job based on the expected performance gains of FPGA acceleration against CPUs. Specifically, we calculate the performance gains by comparing the average execution time of Rosetta real-world applications on the U50 FPGA and CPU, as Rosetta offers both CPU-only and FPGA implementations. As a result, we observe that FPGAs are 1.6× faster than CPUs on average. We apply this speedup factor to the job's execution time. Second, to simulate state management operations, we estimate the FPGA memory usage of each job based on its CPU memory usage. While CPU jobs can offload any portion of their computation to FPGAs in theory, we assume the worst-case scenario for Funky for fairness, where all computations are offloaded to FPGAs, and all FPGA memory buffers need to be saved/restored, leading to the biggest checkpointing overheads. As input/output buffers are duplicated on both CPU and FPGA memory, we assume that FPGA memory usage is always the same as the CPU but limited to its physical memory capacity, i.e., 8 GiB on U50.

Using the modified traces, we implement a full-system orchestration simulator written in Python. The simulator maintains the state of an FPGA-equipped cluster that offers a fixed number of resources (CPUs, memory, vFPGAs) and replays the execution of jobs as they appear in the traces. During the simulation, the simulator parses the traces to retrieve timestamps of scheduling events that happened to tasks: task submission, execution, eviction, failure, and completion. Whenever any events happen, the simulator updates the cluster state and accordingly inserts the Funky-specific overheads, e.g., unikernel bootups.

Scalability. We first evaluate how Funky's performance scales with the number of vFPGAs in the cluster. Assuming that all types of workloads cannot be fully accelerated by FPGAs, we define *acceleration rates*, which indicate the duration tasks can be accelerated with FPGAs during their execution periods. We increase the cluster size from 1 to 128 vFPGAs for five different acceleration rates: 0% (no FPGA acceleration), 25%, 50%, 75%, and 100% (fully accelerated). For each cluster size, we simulate the execution of all tasks in the traces and calculate the throughput, i.e., completed tasks per minute.

Figure 11 shows the results. We observe that an increasing number of vFPGAs and acceleration rates reasonably improve the system throughput. We note that even a small acceleration rate (25%) achieves 1.1× higher throughputs than no FPGA acceleration (0%), indicating that Funky's virtualization overheads are sufficiently small to retain FPGA's performance benefits for real-world traces.

In summary, Funky scales effectively with cluster size on real-world application traces, despite virtualization overheads.

Fault tolerance. We next evaluate the effectiveness of Funky's checkpointing mechanism for fault-tolerant execution. During the simulation, the system periodically takes snapshots of running tasks, while all the tasks fail at random points during their execution (1-99%). Under this setup, the failure events happen at 50% of the total execution time on average, which roughly follows the real-world scenario reported by a prior study [49], i.e., failed cluster jobs run for roughly 40% of their total execution time before the first task failure event. Whenever a task fails, the system resumes the task from the latest snapshot or restarts from the beginning if there are no snapshots. We also examine checkpointing overheads by simulating the case where no failure happens (Success).

Figure 12 shows the average execution times for different checkpointing durations. In any duration, checkpointing reduces the execution time of failed tasks by recovering from the latest snapshot. Regarding the checkpointing duration, frequent checkpointing (e.g., every 30 seconds) increases the performance overheads, while sparse checkpointing (e.g., every 10 minutes) mitigates the overheads in success at the cost of longer recovery time in failure.

In summary, Funky recovers failed task performance using snapshots when the snapshot frequency is optimized.

Scheduling. Lastly, we examine the effectiveness of our preemptive task scheduling for the trace jobs. We evaluate the same scheduling policies used in § 5.5 with 32 vFPGAs.

Figure 13 shows the results, which represent the same trend as Figure 10. Funky's preemptive policies, PRE_EV and PRE_MG, reduce the execution time of high-priority tasks compared to NO_PRE, 5.3% and 4.5% shorter. PRE_MG also reduces 5.9% of the execution time of low-priority tasks compared to PRE_EV because the FPGA is more likely to be occupied for long-running jobs, and task eviction and migration become more effective.

In summary, Funky's preemption mechanism is effective for the production traces as well as the real hardware evaluation.

Solutions	FPGA virtualization and isolation		FPGA state management			Cloud-native orchestration			
	Guest sandbox	FPGA fabric	Guest	Logic	Memory	OCI/CRI	Scheduling	Checkpoint	Scaling
Industry practices									
Cloud instance [6]	VM (standard)	N/A	-	-	-	-	-	-	-
k8s plugin [56, 152]	Container	Vendor's Shell*	-	-	-	✓	✓	-	-
State-of-the-art research									
AmorphOS [81]	User process	Own Shell	-	-	-	-	-	-	-
Coyote [85]	User process	Own Shell	-	-	-	-	-	-	-
SYNERGY [91]	User process	AmorphOS*	-	✓	-	-	-	-	-
Optimus [97]	VM (standard)	Own Shell	-	-	-	-	-	-	-
AvA [161]	VM (standard)	AmorphOS*	✓	✓◇	✓◇	-	-	-	-
BlastFunction [38]	Container	Vendor's Shell*	-	-	-	✓	-	-	✓
Funky	**VM (unikernel)**	Vendor's Shell*	✓	(✓)	✓	✓	✓	✓	✓

*: leverages existing FPGA OSes (Shells) to isolate FPGA fabrics. ◇: adopts *record-and-replay* [102], which does not directly save the states.

Table 7: Comparison with the state-of-the-art approaches for leveraging FPGAs in cloud environments.

6 Related Work

We compare Funky with state-of-the-art research on cloud FPGA management across three dimensions, summarized in Table 7.

FPGA virtualization and isolation. Vendor-provided FPGA platforms adopt the PCIe passthrough that statically binds FPGA devices to guest VMs [6, 103, 143]. While these dedicated instances achieve near-zero FPGA control overheads, the lack of FPGA virtualization leads to low resource utilization in a multi-tenant cloud. FPGA virtualization based on *FPGA OS* applies the OS primitives to hardware tasks on FPGAs [5, 18, 19, 46, 111]. Task schedulers [22, 58, 67, 127, 145], memory virtualization [3, 27, 148], security [76, 86], communication layers [21, 70–72, 101] have been actively studied. AmorphOS [81] and Coyote [85] aim to isolate and share FPGA fabric on a single board rather than cloud-scale distributed FPGA orchestration. Funky leverages such FPGA OSes or vendor-provided Shells [54, 157] to isolate the FPGA fabric and onboard devices (e.g., memory) while it strives to offer a hypervisor-level isolated sandbox for guest CPU applications, which is crucial in a multi-tenant cloud.

Optimus [97] and AvA [161] offer hypervisor-level isolation for standard VMs on traditional cloud FPGA platforms (e.g., Amazon F2 [6]). Although they achieve Funky's security level, they do not consider integration with cloud-native orchestration. In addition, large contexts of standard VMs do not fit cloud-native applications, leading to slow boot time and performance penalties. While AvA provides a VM migration mechanism, integrating it with orchestrators is not its primary goal. In contrast, Funky is the first work to leverage lightweight sandboxes (i.e., unikernels) for FPGA virtualization and integrate the FPGA state management operations into an industry-standard orchestrator, i.e., Kubernetes.

FPGA state management. FPGA context switching [78, 94, 95, 116] and checkpointing [68, 82, 105, 124, 141] have been mainly exploited at the hardware level. They are either based on reading back the state of FPGA logic (registers, BRAMs) or modifying the hardware embedding scan chains to extract the state [68]. SYNERGY [91] is a compiler-based approach that transforms the HDL code of user logic into a preemptible design. While these approaches focus on user logic states, Funky comprehensively breaks down the FPGA workload states in the state-of-the-art architecture (logic, memory, VM) and introduces an end-to-end flow to save and restore them. AvA [161] adopts *record-and-replay* [102], which can induce significant overheads for long-running services.

Cloud-native orchestration. Many orchestration engines have been developed and studied for efficient resource management in data centers and commercial clouds [43, 73, 140], offering preemptive scheduling [24, 77, 121], checkpointing [13, 130], fault tolerance [69], workload scaling [107, 117, 137]. However, they target only CPU and memory resources. GPU orchestration is being rapidly exploited [135, 146, 149] due to the emergence of AI workloads. Funky tackles the orchestration of FPGAs, given their architectural difference from CPUs and GPUs.

There are a few projects to leverage FPGAs in cloud-native environments, which partially address our goals. BlastFunction [38] and F3 [99] integrate FPGAs into Kubernetes-based serverless frameworks, while their functionality is restricted to workload scaling and non-preemptive scheduling. Molecure [47] also supports FPGA acceleration but lacks orchestration and state management support. Unlike these serverless studies, Funky offers comprehensive FPGA virtualization and state management, enabling broader orchestration services like migration, preemption, and checkpointing.

7 Conclusion

We present Funky, an FPGA orchestration engine for cloud-native environments with the following contributions. First, we design *a unikernel architecture for FPGA virtualization* that not only minimizes the performance penalties but also brings faster boot times than containers. We also design a guest OpenCL-compatible library for application portability. Second, we present *a hypervisor-driven FPGA state management*. The thin hypervisor transparently traces host-FPGA data transfers and enables saving and restoring unikernel/FPGA contexts. Lastly, we present three orchestration services: preemptive scheduling, checkpointing, and workload scaling. Our orchestration mechanism is *compatible with the industry-standard CRI/OCI specifications*. We implement and evaluate Funky on our four-node FPGA cluster with three AMD Xilinx FPGAs. Our evaluation demonstrates that Funky imposes virtualization overheads of 7.4% against native execution while enabling FPGA virtualization and state management for orchestration operations.

Artifact availability. The Funky codebase is publicly available at https://github.com/TUM-DSE/Funky.git.

Supplements. Our appendix includes a discussion about Funky's potential applicability for other applications and system domains.

Acknowledgements

This work was supported in parts by an ERC Starting Grant (ID: 101077577) and the Chips Joint Undertaking (JU), European Union (EU) HORIZON-JU-IA, under grant agreement No. 101140087 (SMARTY). The authors acknowledge the financial support by the Federal Ministry of Research, Technology and Space of Germany in the programme of "Souverän. Digital. Vernetzt.". Joint project 6G-life, project identification number: 16KISK002

References

[1] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. Microservice based architecture: Towards high-availability for stateful applications with kubernetes. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, pages 176–185, 2019.

[2] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, page 2–13, New York, NY, USA, 2006. Association for Computing Machinery.

[3] Michael Adler, Kermin E. Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. Leap scratchpads: Automatic memory and cache management for reconfigurable logic. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, page 25–28, New York, NY, USA, 2011. Association for Computing Machinery.

[4] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[5] Andreas Agne, Markus Happe, Ariane Keller, Enno Lübbers, Bernhard Plattner, Marco Platzner, and Christian Plessl. Reconos: An operating system approach for reconfigurable computing. *Micro, IEEE*, 34:60–71, 01 2014.

[6] Amazon. Amazon ec2 f2 instances. https://aws.amazon.com/ec2/instance-types/f2/. Last accessed: October 17, 2025.

[7] AMD. Alveo platforms. https://docs.amd.com/r/en-US/ug1120-alveo-platforms/Alveo-Platforms. Last accessed: October 17, 2025.

[8] Osama G. Attia, Tyler Johnson, Kevin Townsend, Philip Jones, and Joseph Zambreno. Cygraph: A reconfigurable architecture for parallel breadth-first search. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pages 228–235, 2014.

[9] Sameh Attia and Vaughn Betz. Statereveal: Enabling checkpointing of fpga designs with buried state. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pages 206–214, 2020.

[10] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: A java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, page 89–108, New York, NY, USA, 2010. Association for Computing Machinery.

[11] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221, 2012.

[12] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016.

[13] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, page 41–54, New York, NY, USA, 2019. Association for Computing Machinery.

[14] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA, April 2005. USENIX Association.

[15] Thad Benjaponpitak, Meatasit Karakate, and Kunwadee Sripanidkulchai. Enabling live migration of containerized applications across clouds. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, pages 2529–2538, 2020.

[16] Jayaram Bhasker. A vhdl primer. Prentice-Hall, 1992.

[17] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E. Engelstad, and Kyrre Begnum. Includeos: A minimal, resource efficient unikernel for cloud services. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 250–257, 2015.

[18] Gordon Brebner. A virtual hardware operating system for the xilinx xc6200. In Reiner W. Hartenstein and Manfred Glesner, editors, *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*, pages 327–336, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[19] Robert Brodersen, Artem Tkachenko, and Hayden Kwok-Hay So. A unified hardware/software runtime environment for fpga-based reconfigurable computers using borph. In *Proceedings of the 4th International Conference on Hardware/-Software Codesign and System Synthesis (CODES+ISSS '06)*, pages 259–264, 2006.

[20] Jared Casper and Kunle Olukotun. Hardware acceleration of database operations. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '14, page 151–160, New York, NY, USA, 2014. Association for Computing Machinery.

[21] Jiyang Chen, Harshavardhan Unnibhavi, Atsushi Koshiba, and Pramod Bhatotia. vFPIO: A virtual I/O abstraction for FPGA-accelerated I/O devices. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 1167–1184, Santa Clara, CA, July 2024. USENIX Association.

[22] Liang Chen, Thomas Marconi, and Tulika Mitra. Online scheduling for multi-core shared reconfigurable fabric. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 582–585, 2012.

[23] Wei Chen, Jia Rao, and Xiaobo Zhou. Preemptive, low latency datacenter scheduling via lightweight virtualization. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 251–263, Santa Clara, CA, July 2017. USENIX Association.

[24] Long Cheng, Yue Wang, Feng Cheng, Cheng Liu, Zhiming Zhao, and Ying Wang. A deep reinforcement learning-based preemptive approach for cost-aware cloud job scheduling. *IEEE Transactions on Sustainable Computing*, 9(3):422–432, 2024.

[25] Yujeong Choi and Minsoo Rhu. Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 220–233, 2020.

[26] Eric S. Chung, John D. Davis, and Jaewon Lee. Linqits: Big data on little clients. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, page 261–272, New York, NY, USA, 2013. Association for Computing Machinery.

[27] Eric S. Chung, James C. Hoe, and Ken Mai. Coram: An in-fabric memory architecture for fpga-based computing. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, page 97–106, New York, NY, USA, 2011. Association for Computing Machinery.

[28] Alibaba Cloud. Alibaba cloud fpga instances. https://www.alibabacloud.com/help/en/doc-detail/108504.html. Last accessed: October 17, 2025.

[29] Google Cloud. Gpu platforms. https://cloud.google.com/compute/docs/gpus. Last accessed: October 17, 2025.

[30] The Cloud Native Computing Foundation (CNCF). Cncf survey 2020. https://www.cncf.io/wp-content/uploads/2020/12/CNCF_Survey_Report_2020.pdf. Last accessed: October 17, 2025.

[31] Kubernetes community. Container runtime interface (cri). https://github.com/kubernetes/cri-api. Last accessed: October 17, 2025.

[32] Open Container Initiative community. Oci image format specification. https://github.com/opencontainers/image-spec. Last accessed: October 17, 2025.

[33] Open Container Initiative community. Open container initiative runtime specification. https://github.com/opencontainers/runtime-spec. Last accessed: October 17, 2025.

[34] Linux container projects. Lxc - linux containers. https://github.com/lxc/lxc, 2025. Last accessed: October 17, 2025.

[35] NVIDIA Corporation. Nvidia multi-instance gpu. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/, 2024. Last accessed: October 17, 2025.

[36] Vittorio Cozzolino, Oliver Flum, Aaron Yi Ding, and Jörg Ott. Miragemanager: Enabling stateful migration for unikernels. In *Proceedings of the Workshop on Cloud Continuum Services for Smart IoT Systems*, CCIoT '20, page 13–19, New York, NY, USA, 2020. Association for Computing Machinery.

[37] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong 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, page 105–110, New York, NY, USA, 2016. Association for Computing Machinery.

[38] Andrea Damiani, Giorgia Fiscaletti, Marco Bacis, Rolando Brondolin, and Marco D. Santambrogio. Blastfunction: A full-stack framework bringing fpga hardware acceleration to cloud-native applications. *ACM Trans. Reconfigurable Technol. Syst.*, 15(2), jan 2022.

[39] Jonas Dann, Daniel Ritter, and Holger Fröning. Non-relational databases on fpgas: Survey, design decisions, challenges. *ACM Comput. Surv.*, 55(11), feb 2023.

[40] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Kairos: Preemptive data center scheduling without runtime estimates. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 135–148, New York, NY, USA, 2018. Association for Computing Machinery.

[41] Shuiguang Deng, Hailiang Zhao, Binbin Huang, Cheng Zhang, Feiyi Chen, Yinuo Deng, Jianwei Yin, Schahram Dustdar, and Albert Y. Zomaya. Cloud-native computing: A survey from the perspective of services, 2023.

[42] Shuiguang Deng, Hailiang Zhao, Binbin Huang, Cheng Zhang, Feiyi Chen, Yinuo Deng, Jianwei Yin, Schahram Dustdar, and Albert Y. Zomaya. Cloud-native computing: A survey from the perspective of services. *Proceedings of the IEEE*, 112(1):12–46, 2024.

[43] Shuiguang Deng, Hailiang Zhao, Binbin Huang, Cheng Zhang, Feiyi Chen, Yinuo Deng, Jianwei Yin, Schahram Dustdar, and Albert Y. Zomaya. Cloud-native

computing: A survey from the perspective of services. *Proceedings of the IEEE*, 112(1):12–46, 2024.

[44] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. Understanding modern storage apis: a systematic study of libaio, spdk, and io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, SYSTOR '22, page 120–127, New York, NY, USA, 2022. Association for Computing Machinery.

[45] Docker. Swarm mode overview. https://docs.docker.com/engine/swarm/, 2024. Last accessed: October 17, 2025.

[46] Er Domahidi, Eric Chu, and Stephen Boyd. Ecos: An socp solver for embedded systems. In *in European Control Converence*, 2013.

[47] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Serverless computing on heterogeneous computers. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 797–813, New York, NY, USA, 2022. Association for Computing Machinery.

[48] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery.

[49] Nosayba El-Sayed, Hongyu Zhu, and Bianca Schroeder. Learning from failure across multiple clusters: A trace-driven approach to understanding, predicting, and mitigating job terminations. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1333–1344, 2017.

[50] Farah Fahim, Benjamin Hawks, Christian Herwig, James Hirschauer, Sergo Jindariani, Nhan Tran, Luca P. Carloni, Giuseppe Di Guglielmo, Philip Harris, Jeffrey Krupa, Dylan Rankin, Manuel Blanco Valentin, Josiah Hester, Yingyi Luo, John Mamish, Seda Orgrenci-Memik, Thea Aarrestad, Hamza Javed, Vladimir Loncar, Maurizio Pierini, Adrian Alan Pol, Sioni Summers, Javier Duarte, Scott Hauck, Shih-Chieh Hsu, Jennifer Ngadiuba, Mia Liu, Duc Hoang, Edward Kreinar, and Zhenbin Wu. hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices, 2021.

[51] Jian Fang, Yvo T.B. Mulder, Jan Hidders, Jinho Lee, and H. Peter Hofstee. In-memory database acceleration on fpgas: a survey. *VLDB Journal*, 29 (2020)(1):33–59, 2019.

[52] Apache Software Foundation. Apache hadoop yarn, 2025. Last accessed: October 17, 2025.

[53] The Apache Software Foundation. Apache mesos. https://mesos.apache.org/, 2024. Last accessed: October 17, 2025.

[54] Intel FPGA. Accelerator functional unit developer's guide for intel fpga programmable acceleration card. https://www.intel.com/content/www/us/en/programmable/documentation/bfr1522087299048.html. Last accessed: October 17, 2025.

[55] Intel FPGA. Intel fpga add-on for oneapi base toolkit. https://www.intel.com/content/www/us/en/developer/tools/oneapi/fpga.html. Last accessed: October 17, 2025.

[56] Intel FPGA. Intel fpga device plugin for kubernetes. https://github.com/intel/intel-device-plugins-for-kubernetes/tree/main/cmd/fpga_plugin. Last accessed: October 17, 2025.

[57] Intel FPGA. Intel fpga sdk for opencl pro edition: Programming guide. https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807965224.html. Last accessed: October 17, 2025.

[58] Wenyin Fu and Katherine Compton. Scheduling intervals for reconfigurable computing. In *2008 16th International Symposium on Field-Programmable Custom Computing Machines*, pages 87–96, 2008.

[59] Dennis Gannon, Roger Barga, and Neel Sundaresan. Cloud-native applications. *IEEE Cloud Computing*, 4(5):16–21, 2017.

[60] Panagiotis Garefalakis, Konstantinos Karanasos, and Peter Pietzuch. Neptune: Scheduling suspendable tasks for unified stream/batch applications. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 233–245, New York, NY, USA, 2019. Association for Computing Machinery.

[61] Dimitra Giantsidi, Julian Pritzi, Felix Gust, Antonios Katsarakis, Atsushi Koshiba, and Pramod Bhatotia. *TNIC: A Trusted NIC Architecture: A hardware-network substrate for building high-performance trustworthy distributed systems*, page 1282–1301. Association for Computing Machinery, New York, NY, USA, 2025.

[62] Google. Cloud tensor processing units (tpus). https://cloud.google.com/tpu, 2024. Last accessed: October 17, 2025.

[63] Google. Clusterdata 2019 traces. https://github.com/google/cluster-data/blob/master/ClusterData2019.md, 2024. Last accessed: October 17, 2025.

[64] Khronos Group. Opencl api c++ bindings. https://github.com/KhronosGroup/OpenCL-CLHPP. Last accessed: October 17, 2025.

[65] Khronos Group. Opencl api headers. https://github.com/KhronosGroup/OpenCL-Headers. Last accessed: October 17, 2025.

[66] Khronos Group. The opencl specification. https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html. Last accessed: October 17, 2025.

[67] Brandon Kyle Hamilton, Michael Inggs, and Hayden Kwok Hay So. Scheduling mixed-architecture processes in tightly coupled fpga-cpu reconfigurable computers. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 240–240, 2014.

[68] Markus Happe, Andreas Traber, and Ariane Keller. Preemptive hardware multitasking in reconos. In *ARC*, 2015.

[69] Moin Hasan and Major Singh Goraya. Fault tolerance in cloud computing environment: A systematic survey. *Computers in Industry*, 99:156–172, 2018.

[70] Matthias Hille, Nils Asmussen, Pramod Bhatotia, and Hermann Härtig. SemperOS: A distributed capability system. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 709–722, Renton, WA, July 2019. USENIX Association.

[71] Matthias Hille, Nils Asmussen, Hermann Härtig, and Pramod Bhatotia. A heterogeneous microkernel os for rack-scale systems. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '20, page 50–58, New York, NY, USA, 2020. Association for Computing Machinery.

[72] Chun-Hsian Huang and Pao-Ann Hsiung. Hardware resource virtualization for dynamically partially reconfigurable systems. *IEEE Embedded Systems Letters*, 1(1):19–23, 2009.

[73] Google Inc. Kubernetes (k8s). https://github.com/kubernetes/kubernetes. Last accessed: October 17, 2025.

[74] Google Inc. Scheduling, preemption and eviction on kubernetes. https://kubernetes.io/docs/concepts/scheduling-eviction. Last accessed: October 17, 2025.

[75] Intel. Open fpga stack overview. https://ofs.github.io/ofs-2024.2-1/. Last accessed: October 17, 2025.

[76] Chenglu Jin, Vasudev Gohil, Ramesh Karri, and Jeyavijayan Rajendran. Security of cloud fpgas: A survey, 2020.

[77] JCS Kadupitige, Vikram Jadhao, and Prateek Sharma. Modeling the temporally constrained preemptions of transient cloud vms. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '20, page 41–52, New York, NY, USA, 2020. Association for Computing Machinery.

[78] H. Kalte and M. Porrmann. Context saving and restoring for multitasking in reconfigurable systems. In *International Conference on Field Programmable Logic and Applications, 2005.*, pages 223–228, 2005.

[79] Kiranpreet Kaur, Fabrice Guillemin, and Francoise Sailhan. Container placement and migration strategies for cloud, fog, and edge data centers: A survey. *International Journal of Network Management*, 32(6), November 2022.

[80] Kiranpreet Kaur, Fabrice Guillemin, and Francoise Sailhan. Live migration of containerized microservices between remote Kubernetes Clusters. In *IEEE International Conference on Computer Communications - Workshop (INFOCOM)*, pages 114–119, New York, United States, May 2023.

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

[82] Oliver Knodel, Paul R. Genssler, and Rainer G. Spallek. Migration of long-running tasks between reconfigurable resources using virtualization. *SIGARCH Comput. Archit. News*, 44(4):56–61, jan 2017.

[83] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: A language and compiler for application accelerators. *SIGPLAN Not.*, 53(4):296–311, jun 2018.

[84] David Koeplinger, Raghu Prabhakar, Yaqi Zhang, Christina Delimitrou, Christos Kozyrakis, and Kunle Olukotun. Automatic generation of efficient accelerators for reconfigurable hardware. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 115–127, 2016.

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

[86] Atsushi Koshiba, Felix Gust, Julian Pritzi, Anjo Vahldiek-Oberwagner, Nuno Santos, and Pramod Bhatotia. Trusted heterogeneous disaggregated architectures. In *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '23, page 72–79, New York, NY, USA, 2023. Association for Computing Machinery.

[87] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: Fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 376–394, New York, NY, USA, 2021. Association for Computing Machinery.

[88] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. A linux in unikernel clothing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[89] Dmitrii Kuvaiskii, Dimitrios Stavrakakis, Kailun Qin, Cedric Xing, Pramod Bhatotia, and Mona Vij. Gramine-tdx: A lightweight os kernel for confidential vms. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 4598–4612, New York, NY, USA, 2024.

Association for Computing Machinery.

[90] Dongup Kwon, Junehyuk Boo, Dongryeong Kim, and Jangwoo 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)*, pages 955–971. USENIX Association, November 2020.

[91] Joshua Landgraf, Tiffany Yang, Will Lin, Christopher J. Rossbach, and Eric Schkufza. Compiler-driven fpga virtualization with synergy. *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[92] J. Lebak, J. Kepner, H. Hoffmann, and E. Rutledge. Parallel vsipl++: An open standard software library for high-performance parallel signal processing. *Proceedings of the IEEE*, 93(2):313–330, 2005.

[93] Ilia Lebedev, Christopher Fletcher, Shaoyi Cheng, James Martin, Austin Doupnik, Daniel Burke, Mingjie Lin, and John Wawrzynek. Exploring many-core design templates for fpgas and asics. *Int. J. Reconfig. Comput.*, 2012, jan 2012.

[94] Trong-Yen Lee, Che-Cheng Hu, Li-Wen Lai, and Chia-Chun Tsai. Hardware context-switch methodology for dynamically partially reconfigurable systems. *J. Inf. Sci. Eng.*, 26:1289–1305, 2010.

[95] L. Levinson, R. Manner, M. Sessler, and H. Simmler. Preemptive multitasking on fpgas. In *Proceedings 2000 IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No.PR00871)*, pages 301–302, 2000.

[96] Alec Lu and Zhenman Fang. Sql2fpga: Automatic acceleration of sql query processing on modern cpu-fpga platforms. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 184–194, 2023.

[97] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. A hypervisor for shared-memory fpga platforms. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 827–844, New York, NY, USA, 2020. Association for Computing Machinery.

[98] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *SIGARCH Comput. Archit. News*, 41(1):461–472, mar 2013.

[99] Charalampos Mainas, Martin Lambeck, Bruno Scheufler, Laurent Bindschaedler, Atsushi Koshiba, and Pramod Bhatotia. F3: An fpga-accelerated faas framework. In *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '25, New York, NY, USA, 2025. Association for Computing Machinery.

[100] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery.

[101] Joel Mandebi Mbongue, Festus Hategekimana, Danielle Tchuinkou Kwadjo, and Christophe Bobda. Fpga virtualization in cloud-based infrastructures over virtio. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 242–245, 2018.

[102] Violeta Medina and Juan Manuel García. A survey of migration mechanisms of virtual machines. *ACM Comput. Surv.*, 46(3), jan 2014.

[103] Microsoft. Np size series - azure virtual machines. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/fpga-accelerated/np-series, 2025. Last accessed: October 17, 2025.

[104] Masanori Misono, Peter Okelmann, Charalampos Mainas, and Pramod Bhatotia. uio: Lightweight and extensible unikernels. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, SoCC '24, page 580–599, New York, NY, USA, 2024. Association for Computing Machinery.

[105] Aurelio Morales-Villanueva and Ann Gordon-Ross. Partial region and bitstream cost models for hardware multitasking on partially reconfigurable fpgas. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 90–96, 2015.

[106] Fahad Bin Muslim, Liang Ma, Mehdi Roozmeh, and Luciano Lavagno. Efficient fpga implementation of opencl high-performance computing applications via high-level synthesis. *IEEE Access*, 5:2747–2762, 2017.

[107] Thanh-Tung Nguyen, Yu-Jin Yeom, Taehong Kim, Dae-Heon Park, and Sehan Kim. Horizontal pod autoscaling in kubernetes for elastic container orchestration. *Sensors*, 20(16), 2020.

[108] Nubificus. Kata containers - open source container runtime software. https://katacontainers.io/. Last accessed: October 17, 2025.

[109] Tayo Oguntebi and Kunle 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, page 111–117, New York, NY, USA, 2016. Association for Computing Machinery.

[110] Data on Kubernetes Community. Data on kubernetes 2021 report. https://dok.community/dokc-2021-report/. Last accessed: October 17, 2025.

[111] Wesley Peck, Erik Anderson, Jason Agron, Jim Stevens, Fabrice Baijot, and David Andrews. Hthreads: A computational model for reconfigurable devices. In *2006 International Conference on Field Programmable Logic and Applications*, pages 1–4, 2006.

[112] Tobias Pfandzelter, Aditya Dhakal, Eitan Frachtenberg, Sai Rahul Chalamalasetti, Darel Emmot, Ninad Hogade, Rolando Pablo Hong Enriquez, Gourav Rattihalli, David Bermbach, and Dejan Milojicic. Kernel-as-a-service: A serverless programming model for heterogeneous hardware accelerators. In *Proceedings of the 24th International Middleware Conference*, Middleware '23, page 192–206, New York, NY, USA, 2023. Association for Computing Machinery.

[113] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *IEEE Micro*, 35(3):10–22, 2015.

[114] Giovanni Quattrocchi, Emilio Incerto, Riccardo Pinciroli, Catia Trubiani, and Luciano Baresi. Autoscaling solutions for cloud applications under dynamic workloads. *IEEE Transactions on Services Computing*, 17(3):804–820, 2024.

[115] Masudul Hassan Quraishi, Erfan Bank Tavakoli, and Fengbo Ren. A survey of system architectures and techniques for fpga virtualization, 2021.

[116] Kyle Rupnow, Wenyin Fu, and Katherine Compton. Block, drop or roll(back): Alternative preemption methods for rh multi-tasking. In *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, pages 63–70, 2009.

[117] Lluís Mas Ruíz, Pere Piñol Pueyo, Jordi Mateo-Fornés, Jordi Vilaplana Mayoral, and Francesc Solsona Tehàs. Autoscaling pods on an on-premise kubernetes infrastructure qos-aware. *IEEE Access*, 10:33083–33094, 2022.

[118] Patrick Sabanic, Masanori Misono, Teofil Bodea, Julian Pritzi, Michael Hackl, Dimitrios Stavrakakis, and Pramod Bhatotia. Confidential serverless computing, 2025.

[119] R. Sass, D. Andrews, E. Komp, W. Peck, F. Baijot, E. Anderson, J. Stevens, and J. Agron. Enabling a uniform programming model across the software/hardware boundary. In *2006 14th Annual IEEE Symposium on Field Programmable Custom Computing Machines*, pages 89–98, Los Alamitos, CA, USA, apr 2006. IEEE Computer Society.

[120] Henri Schmidt, Zeineb Rejiba, Raphael Eidenbenz, and Klaus-Tycho Förster. Transparent fault tolerance for stateful applications in kubernetes with checkpoint/restore. In *2023 42nd International Symposium on Reliable Distributed Systems (SRDS)*, pages 129–139, 2023.

[121] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 351–364, New York, NY, USA, 2013. Association for Computing Machinery.

[122] Ahmad Shawahna, Sadiq M. Sait, and Aiman El-Maleh. Fpga-based accelerators of deep learning networks for learning and classification: A review. *IEEE Access*, 7:7823–7859, 2019.

[123] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 121–135, New York, NY, USA, 2019. Association for Computing Machinery.

[124] H. Simmler, L. Levinson, and Reinhard Männer. Multitasking on fpga coprocessors. In *Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications*, FPL '00, page 121–130, Berlin, Heidelberg, 2000. Springer-Verlag.

[125] Gursharan Singh, Parminder Singh, Anas Motii, and Mustapha Hedabou. A secure and lightweight container migration technique in cloud computing. *Journal of King Saud University - Computer and Information Sciences*, 36(1):101887, 2024.

[126] Piush K Sinha, Spoorti S Doddamani, Hui Lu, and Kartik Gopalan. mwarp: Accelerating intra-host live container migration via memory warping. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 508–513, 2019.

[127] C. Steiger, H. Walder, and M. Platzner. Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. *IEEE Transactions on Computers*, 53(11):1393–1407, 2004.

[128] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu 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, page 16–25, New York, NY, USA, 2016. Association for Computing Machinery.

[129] Márk Szalay, Péter Mátray, and László Toka. State management for cloud-native applications. *Electronics*, 10(4), 2021.

[130] Márk Szalay, Péter Mátray, and László Toka. State management for cloud-native applications. *Electronics*, 10(4), 2021.

[131] Ariel Szekely, Adam Belay, Robert Morris, and M. Frans Kaashoek. Unifying serverless and microservice workloads with sigmaos. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 385–402, New York, NY, USA, 2024. Association for Computing Machinery.

[132] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Cntr: Lightweight OS containers. In *2018 USENIX Annual Technical Conference*

(USENIX ATC 18), pages 199–212, Boston, MA, July 2018. USENIX Association.

[133] Jörg Thalheim, Peter Okelmann, Harshavardhan Unnibhavi, Redha Gouicem, and Pramod Bhatotia. Vmsh: hypervisor-agnostic guest overlays for vms. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 678–696, New York, NY, USA, 2022. Association for Computing Machinery.

[134] Jörg Thalheim, Harshavardhan Unnibhavi, Christian Priebe, Pramod Bhatotia, and Peter Pietzuch. rkt-io: a direct i/o stack for shielded execution. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 490–506, New York, NY, USA, 2021. Association for Computing Machinery.

[135] Prashanth Thinakaran, Jashwant Raj Gunasekaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R. Das. Kube-knots: Resource harvesting through dynamic container orchestration in gpu-based datacenters. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–13, 2019.

[136] Donald Thomas and Philip Moorby. The verilog hardware description language. Springer Science & Business Media, 1991.

[137] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[138] Minh-Ngoc Tran, Xuan Tuong Vu, and Younghan Kim. Proactive stateful fault-tolerant system for kubernetes containerized services. *IEEE Access*, 10:102181–102194, 2022.

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

[140] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[141] Hoang-Gia Vu, Takashi Nakada, and Yasuhiko Nakashima. Efficient hardware task migration for heterogeneous fpga computing using hdl-based checkpointing. *Integration*, 77, 12 2020.

[142] Dong Wang, Ke Xu, and Diankun Jiang. Pipecnn: An opencl-based open-source fpga accelerator for convolution neural networks. In *2017 International Conference on Field Programmable Technology (ICFPT)*, pages 279–282, 2017.

[143] Xiuxiu Wang, Yipei Niu, Fangming Liu, and Zichen Xu. When fpga meets cloud: A first look at performance. *IEEE Transactions on Cloud Computing*, 10(2):1344–1357, 2022.

[144] Zeke Wang, Johns Paul, Hui Yan Cheah, Bingsheng He, and Wei Zhang. Relational query processing on opencl-based fpgas. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–10, 2016.

[145] Guy Wassi, Mohamed El Amine Benkhelifa, Geoff Lawday, François Verdier, and Samuel Garcia. Multi-shape tasks scheduling for online multitasking on fpgas. In *2014 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–7, 2014.

[146] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. Beware of fragmentation: Scheduling GPU-Sharing workloads with fragmentation gradient descent. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 995–1008, Boston, MA, July 2023. USENIX Association.

[147] Dan Williams and Ricardo Koller. Unikernel monitors: Extending minimalism outside of the box. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'16, page 71–76, USA, 2016. USENIX Association.

[148] Felix Winterstein, Kermin Fleming, Hsin-Jung Yang, Samuel Bayliss, and George Constantinides. Matchup: Memory abstractions for heap manipulating programs. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, page 136–145, New York, NY, USA, 2015. Association for Computing Machinery.

[149] Hao Wu, Yue Yu, Junxiao Deng, Shadi Ibrahim, Song Wu, Hao Fan, Ziyue Cheng, and Hai Jin. StreamBox: A lightweight GPU SandBox for serverless inference workflow. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 59–73, Santa Clara, CA, July 2024. USENIX Association.

[150] Shucai Xiao, Pavan Balaji, Qian Zhu, Rajeev Thakur, Susan Coghlan, Heshan Lin, Gaojin Wen, Jue Hong, and Wu-chun Feng. Vocl: An optimized environment for transparent virtualization of graphics processing units. In *2012 Innovative Parallel Computing (InPar)*, pages 1–12, 2012.

[151] AMD Xilinx. Alveo u50 data center accelerator card. https://www.xilinx.com/products/boards-and-kits/alveo/u50.html. Last accessed: October 17, 2025.

[152] AMD Xilinx. Fpga as a service. https://github.com/Xilinx/FPGA_as_a_Service. Last accessed: October 17, 2025.

[153] AMD Xilinx. Vitis accel examples. https://github.com/Xilinx/Vitis_Accel_Examples. Last accessed: October 17, 2025.

[154] AMD Xilinx. Vivado ml edition. https://www.xilinx.com/products/design-tools/vivado.html. Last accessed: October 17, 2025.

[155] AMD Xilinx. Xilinx base runtime. https://github.com/Xilinx/Xilinx_Base_Runtime. Last accessed: October 17, 2025.

[156] AMD Xilinx. Xilinx opencl extension. https://xilinx.github.io/XRT/master/html/opencl_extension.html. Last accessed: October 17, 2025.

[157] AMD Xilinx. Xrt and vitis platform overview. https://xilinx.github.io/XRT/master/html/platforms.html. Last accessed: October 17, 2025.

[158] AMD Xilinx. Xrt native apis. https://xilinx.github.io/XRT/master/html/xrt_native_apis.html. Last accessed: October 17, 2025.

[159] Bo Xu, Song Wu, Jiang Xiao, Hai Jin, Yingxi Zhang, Guoqiang Shi, Tingyu Lin, Jia Rao, Li Yi, and Jizhong Jiang. Sledge: Towards efficient live migration of docker containers. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 321–328, 2020.

[160] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The true cost of containing: A gVisor case study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[161] Hangchen Yu, Arthur Michener Peters, Amogh Akshintala, and Christopher J. Rossbach. Ava: Accelerated virtualization of accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 807–825, New York, NY, USA, 2020. Association for Computing Machinery.

[162] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason 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, page 161–170, New York, NY, USA, 2015. Association for Computing Machinery.

[163] Jialiang Zhang and Jing Li. Improving the performance of opencl-based fpga accelerator for convolutional neural network. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 25–34, New York, NY, USA, 2017. Association for Computing Machinery.

[164] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, and Zhiru Zhang. Rosetta: A realistic high-level synthesis benchmark suite for software programmable fpgas. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, page 269–278, New York, NY, USA, 2018. Association for Computing Machinery.