ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

vFPIO: A Virtual I/O Abstraction for FPGA-accelerated I/O Devices

Jiyang Chen, Harshavardhan Unnibhavi, Atsushi Koshiba, Pramod Bhatotia
Technical University of Munich

Abstract

Modern cloud systems have adopted a variety of FPGA-accelerated I/O devices, such as SmartNICs and computational storage, while they face programmability and portability challenges. Existing FPGA frameworks either directly expose device-specific I/O interfaces to user logic or offer virtualized I/Os limited to a single device type. The lack of I/O abstraction imposes high engineering costs, less design portability, and even unexpected throughput degradation.

We introduce vFPIO, an FPGA-based I/O acceleration framework that brings better programmability and design portability. vFPIO extends modern FPGA OSes to expose *virtual I/O ports* to user logic, which abstracts device-dependent I/O specifications and makes the user logic design platform-agnostic. The connectivity between virtual and physical I/O ports can be easily configured by host applications using POSIX-like file APIs. vFPIO also offers a preemptive I/O transaction scheduler that alleviates the I/O throughput degradation caused by concurrent I/O requests from multiple accelerators in a multi-tenant environment.

We implement a prototype of the vFPIO framework on x86 servers equipped with AMD Xilinx Alveo U280 cards and support four different I/O interfaces: PCIe, DRAM, HBM, and network. Our evaluation highlights that vFPIO incurs negligible performance overheads compared to Coyote, one of the latest FPGA OSes, while preserving the maximum I/O throughput for high-priority tasks even under resource contention.

1 Introduction

I/O acceleration using Field-Programmable Gate Arrays (FPGAs) is an emerging trend in modern cloud computing systems. Network and storage interfaces have rapidly evolved to improve the I/O bandwidth (e.g., Terabit Ethernet [20], PCIe 6.0 [11]) and high-speed data protocols (e.g., NVMe [9], CXL [6]). Although these high-speed I/Os contribute to accelerating data-intensive workloads, traditional CPU-centric architectures cannot fully exploit them because of redundant data movements between CPUs and devices [48, 55]. To bridge this gap, the industry and research groups leverage FPGAs to manage high-speed I/O devices without host CPU intervention: PCIe-based FPGA cards [1, 7], SmartNICs [2,8,27], computational storages [12,46,47,55,56]. These smart devices allow custom accelerators on FPGAs to directly process data streams transferred through the high-speed I/Os, which achieves stable and massive performance gains [30].

Despite the high performance and flexibility of FPGA-driven I/Os, designing and developing FPGA accelerators managing various I/Os impose high engineering efforts due to the lack of I/O abstraction. FPGA vendors offer preset hardware modules configured on a static region to facilitate connecting user logic and external I/O devices (i.e., *Shells*). However, they expose dedicated I/O interfaces to a programmable region and leave it to user logic to control device-specific I/Os [5, 17] or pose limitations to supported I/O devices [10]. Consequently, developers need to carefully design their user logic to combine computation logic and I/O control logic. Such complex hardware design burdens developers and leads to poor portability because the computational logic of FPGA accelerators is tightly coupled with device-dependent I/O logic.

While recent studies have explored FPGA operating systems for abstracting both on-chip and off-chip peripherals from user logic [32, 34, 36], challenges persist in supporting a broader range of I/O devices and effectively managing I/O contention. The state-of-the-art FPGA OSes focus on spatial sharing and memory virtualization, allowing multiple accelerators to be configured on a single FPGA chip while preserving memory isolation. However, they fall short in I/O abstraction; some only target memory I/Os [32, 36], while others expose device-specific I/O interfaces [34]. Moreover, existing spatial sharing mechanisms are susceptible to I/O contention, where multiple accelerators simultaneously request access to the same I/O interface, causing performance degradation. Prior approaches like Coyote and FSRF [34, 36] distribute available bandwidth fairly among accelerators using a round-robin scheme. This method cannot prioritize I/O requests, which is critical for latency- and throughput-sensitive workloads.

These challenges motivate us to *design a new FPGA I/O acceleration framework that abstracts device-specific I/O interfaces from users' accelerators and improves programmability and portability while mitigating I/O contentions among multiple accelerators.*

We propose **vFPIO**, an FPGA-based I/O acceleration framework that simplifies FPGA development with three key features. First, vFPIO abstracts user logic I/O ports as *virtual I/Os*. These virtual I/Os act as a layer of abstraction, decoupling device-specific I/O control logic from user logic and allowing the user logic to connect dynamically to various external I/O devices without code modification. Second, vFPIO offers *vFPIO APIs*, POSIX-like file APIs that allow host applications to configure virtual I/O connections without reconfiguring FPGA. These simple APIs reduce the development efforts of FPGA I/O applications. Third, vFPIO offers an FPGA Shell equipped with *I/O transaction schedulers* that reorder I/O transaction requests to corresponding DMAs and I/O controllers. The schedulers allow I/O-intensive workloads to preserve the maximum throughput regardless of the other accelerators running in parallel.

We implement the vFPIO framework on x86 servers equipped with Alveo U280, a datacenter-grade AMD Xilinx FPGA card. The key hardware components of vFPIO are implemented by extending Coyote [34], an open-source FPGA OS. Our implementation supports two types of memory devices (host DRAM and FPGA HBM), PCIe, and network (RDMA) for I/O acceleration.

We evaluate vFPIO's programmability, portability, and performance isolation with various FPGA applications. Our evaluation results highlight that vFPIO incurs only 1% performance overheads on average for the application's end-to-end performance compared to Coyote. vFPIO also ensures consistent I/O throughput of high-priority applications regardless of co-running applications.

Our main contributions are as follows:

- **FPGA I/O virtualization.** The virtual I/O ports allow developers to focus on designing the computation part only while improving the portability of user logic for various FPGA-accelerated devices.
- **Dynamic I/O switching between devices.** We design POSIX-like file APIs that allow applications to dynamically switch connections among different devices without time-consuming FPGA reconfiguration.
- **I/O performance isolation.** We propose an I/O transaction scheduler that reorders data transfer requests from user logic according to user-defined priorities. It guarantees the throughput of I/O-critical workloads even if the I/O contention happens.

2 Background and Motivation

2.1 Background

FPGA-driven I/O. FPGAs are more suitable for stream data processing than CPUs as they are designed to execute multiple operations in parallel and can better pipeline operations. They can be configured to implement custom hardware logic for the target algorithm to improve throughput, reduce CPU usage, and save energy consumption. They have been deployed for near-data processing, such as near-memory [51] and near-storage [46] computing. FPGAs can also include a network stack for high-throughput packet processing [29].

FPGA programming. FPGA applications are composed of two parts: the host code and the FPGA kernel. The host code runs on CPUs and communicates with FPGA for initialization and task offloading. The kernel represents the user logic configured on FPGA and is programmed using hardware description languages (HDL) such as Verilog and VHDL. High-Level Synthesis (HLS) has evolved to simplify this development flow by allowing developers to use high-level languages such as C and C++ to describe the hardware design [40].

FPGA Shell and OS. The FPGA Shell is a set of I/O and control logic programmed in the static region of the FPGA. The Shells provided by FPGA vendors typically offer communication interfaces between host (CPU) and accelerators, interconnects to onboard devices (DDR, network ports), and DMAs [16]. Some FPGA OSes have been proposed as a set of dedicated hardware modules integrated into the Shell [32, 34, 36, 57]. They are designed to allow multiple accelerators to be configured on the same FPGA and offer OS-level resource abstractions, such as spatial and temporal multiplexing [32, 34] and memory abstraction [34, 36].

2.2 Motivation

State-of-the-art. FPGA developers commonly use vendor-provided Shells for designing and building I/O accelerators on modern FPGA devices [5, 16, 17]. These vendor-provided Shells implement dedicated controllers for external I/Os and expose low-level interfaces to reconfigurable slots where user logic is programmed. Using these Shells, however, developers must carefully design their user logic so that its I/O specifications (data protocol types, port widths, and the number of I/O ports) meet the target I/Os requirements. Moreover, even minor adjustments to the I/O configuration, such as connecting a new device, demand an update of the entire user logic design.

Meanwhile, many FPGA OSes have been proposed for multi-tenancy support on FPGA [32, 34, 36, 38, 57], which support I/O abstraction. However, their features mainly focus on memory virtualization and need to be more for various I/O devices. AmorphOS [32] and Coyote [34] support virtual memory, abstracting local FPGA memories and translating memory addresses accessed from user logic. FSRF [36] supports file system-based memory abstraction, which allows host applications to map files into virtual FPGA memories via POSIX-like APIs. While these FPGA OSes handle off-chip memory (DRAM, HBM) abstraction, other I/O devices remain unaddressed. Although Coyote [34] provides network interfaces, it directly exposes them to user logic without virtualization.

Limitations. Despite progress in I/O abstraction, state-of-the-art FPGA systems still face limitations hindering developer adoption.

First, they do not fully abstract device-specific I/O interfaces from the hardware design of user logic. Existing FPGA OSes [32, 34, 36] remain tethered to specific I/O details and force developers to work with low-level I/O control details, such as interface protocol, port level protocol, and register mode, to ensure proper connection and optimal data throughput. Consequently, user logic designs become burdened with device-specific I/O control functionalities.

Second, they are limited in the range of integrated I/O interfaces. Some accelerators designed for generic algorithms, such as hashing and data compression, can be adapted to different types of I/O devices; they can process data regardless of the format and potentially work with diverse I/O sources, e.g., both disk and network I/Os. However, due to the limitations of FPGA programming, they must be re-synthesized and compiled when I/O interfaces are modified. This process can take hours or even days [26,35,64], significantly impacting development time. While user logic can leverage the same interfaces for various back-end I/O devices as data sources or sinks, existing frameworks hinder efficient switching between these sources and sinks.

Third, they lack mechanisms to handle I/O contentions when multiple accelerators compete for the same interface. Existing FPGA OSes simply multiplex I/O requests and fairly divide the available bandwidth for accelerators using the same I/O device, i.e., round robin [32, 34, 38]. The growing popularity of frameworks supporting temporal sharing of FPGA between diverse FPGA applications exposes I/O contention as a critical performance bottleneck. When multiple user logics run concurrently using the same I/O device, their throughput suffers significant degradation.

Our proposal. To address these challenges, we propose the vFPIO framework, a hardware-software co-design that offers improved programmability and portability for FPGA I/O accelerators while ensuring the I/O throughput for high-priority tasks during resource contention. vFPIO thrives in dynamic environments like multi-tenant clouds where multiple applications with varying priorities compete for a limited pool of FPGA devices; vFPIO enables efficient time-sharing through fast dynamic reconfiguration, and its priority scheduler gives critical workloads better access to hardware resources than non-critical ones.

3 Overview

3.1 System Overview

The vFPIO framework simplifies coding, enhances flexibility for various I/O devices, and handles I/O contention in a multi-tenant situation with the help of the virtual I/O concept. The key insight is that vFPIO exposes virtual I/O ports to user logic, eliminating the need for direct connections to devices and letting the underlying hardware/software components transparently handle the actual I/O connections. This approach decouples device-specific I/O interfaces from users' hardware design and allows host applications to configure these devices using a unified interface.

System components. Figure 1 illustrates a system overview of the vFPIO framework. We assume that the FPGA device is connected to the host machine through PCI Express as well as modern FPGA devices such as cloud FPGA cards [1], SmartNICs [2], and SmartSSDs [12]. The FPGA board offers one or more reconfigurable slots named virtual FPGA (vFPGA), where user logic is configured. The vFPIO framework comprises three main components: vFPIO library, virtual I/O module, and scheduler. *The vFPIO library* offers POSIX-like file APIs to configure I/O connections between user logic and devices and initiate I/O transactions. *The Virtual I/O module* abstracts platform-specific I/O ports from user logic and sets up the interfaces between vFPGA and I/O devices. *The scheduler* is responsible for scheduling I/O transactions.

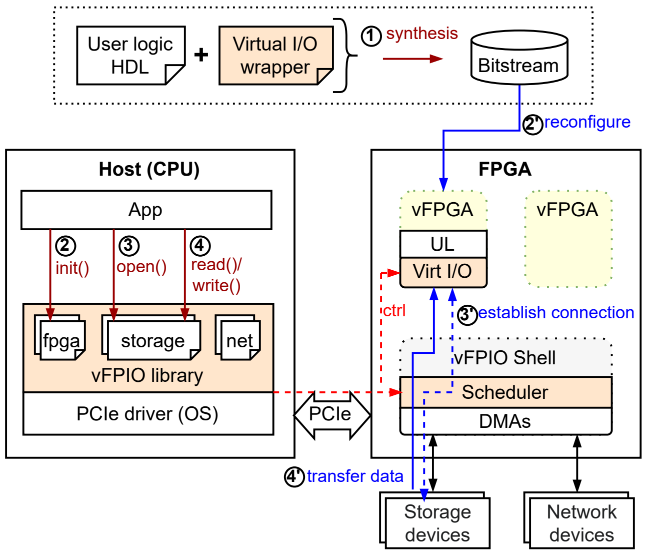

Figure 1: An overview of the vFPIO framework.

System workflow. The HDL code of user logic must be synthesized with the virtual I/O wrapper. The wrapper contains hardware modules exposing virtual I/O ports to user logic (step ①). At the execution phase, the host application initializes vFPGAs with the bitstream of user logic (step ②) and then configures the connection between the user logic and external I/Os via POSIX-like file APIs such as open() and close() (step ③). These APIs write parameters to memory-mapped registers of the scheduler and the Virtual I/O module. Lastly, data transfers are triggered by read() and write() APIs (step ④). These APIs invoke DMAs dedicated to managing the external I/Os.

Design goals. We design the vFPIO framework to achieve the following goals.

- **Programmability**: vFPIO allows developers to design their user logic without being aware of I/O constraints such as port widths and device types, which brings better application programmability and compatibility.
- **Portability**: vFPIO allows applications to dynamically switch the interconnects between user logic and external I/Os without changing the hardware design or reconfiguring the FPGA.
- **Performance isolation**: vFPIO alleviates I/O contentions caused when multiple accelerators invoke I/O transactions in parallel to retain the peak I/O throughput for high-priority applications.

3.2 Design Challenges and Key Ideas

vFPIO addresses the following design challenges.

#1 Virtualizing device-specific I/O interfaces. For FPGA I/O acceleration, user logic connects to external I/O ports offered by FPGA Shells, which imposes a design limitation.

Existing Shells and FPGA OSes are equipped with dedicated I/O controllers and DMAs that convert device-specific interfaces (e.g., PCIe, QSFP, DDR) to common I/O protocols (e.g., AXI stream) and expose them to user logic [5, 34]. However, the Shell-provided I/O ports still impose constraints on the user logic design, such as the number of I/O ports and bus widths. These constraints make the user logic design incompatible among different FPGA platforms and not connectable to different I/O devices supported by the same Shell without updating the design.

We propose a virtual I/O module, a wrapper module that abstracts external (*physical*) I/O ports as *virtual I/O ports*. The module allows user logic to be device-agnostic and connect to any I/O device regardless of device type, improving the design portability and compatibility (described in § 4.2).

#2 Dynamic FPGA I/O configuration. Programming interfaces to configure virtual I/O modules must be exposed to host applications so that applications flexibly change I/O directions of user logic across various I/O devices on demand. However, modern FPGA systems relying on Shells and OSes [16, 34, 36] do not offer such flexible I/O programming models because they statically configure I/O interconnects between user logic and devices. BORCH [53] and FSRF [36] explore a file-based abstraction and file I/O APIs for memory devices, while they cannot cover different I/O interfaces such as networks.

To solve this challenge, the vFPIO framework provides vFPIO APIs, POSIX-like file APIs that allow FPGA applications to dynamically change the interconnects between user logic and I/O devices (described in § 4.3).

#3 Device I/O contention. Current FPGA frameworks that support multiple vFPGA slots offer only a Round-Robin policy for I/O transaction scheduling [32, 34, 36]. This scheduling policy does not solve resource contention issues when many user logics use the same I/O device. Implementing a preemptive priority-based scheduling policy requires the ability to pause the execution of low-priority applications and resume later without errors.

We provide an I/O scheduler that supports a preemptive priority-based scheduling policy. In resource contention, high-priority user logic can preempt others and execute first. This design guarantees that critical application execution is not affected by other co-existing applications (described in § 4.4).

4 Design

4.1 vFPIO Shell

Figure 2 illustrates the detailed architecture of the vFPIO Shell. The vFPIO Shell adopts a streaming channel for data buses between user logic and dedicated I/O controllers/DMAs. The Shell also uses a lightweight control interface for control buses. As well as the latest FPGA OSes [32, 34, 36], the vFPIO Shell is equipped with two types of fundamental hardware modules: MMUs for memory virtualization and dedicated I/O controllers/DMAs for individual I/O devices. The MMUs oversee the address of host/local memories and manage a virtual

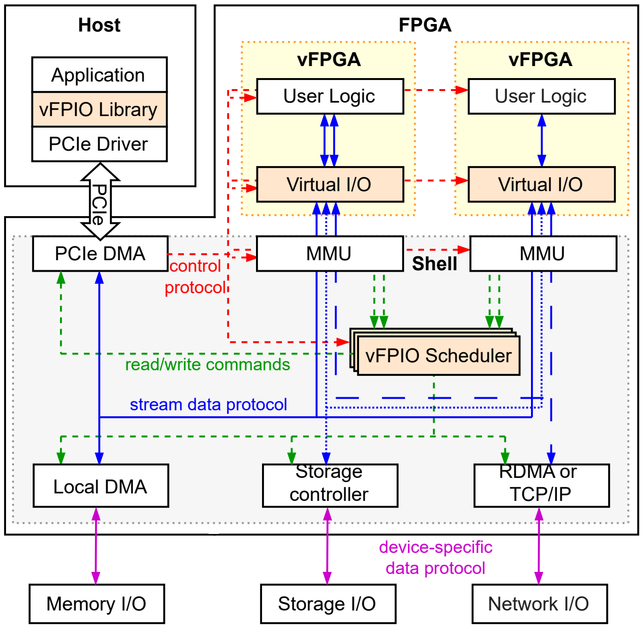

Figure 2: vFPIO Shell architecture. The architecture is based on Coyote [34], one of the latest FPGA OS. The key components are colored orange to highlight the changes from Coyote [34].

memory space provided to user logic. They translate virtual and physical memory addresses of read/write requests when handling memory reads/writes and invoke the corresponding DMAs to trigger data transfers. Upon the DMA requests issued by the MMU, the corresponding controller or DMA invokes stream data transfers between the user logic and the device. The DMAs and controllers are also responsible for converting the streaming protocol into device-specific data protocols.

We highlight two hardware modules in the Shell architecture: the *virtual I/O modules* and the *vFPIO scheduler*. We assume host applications trigger all data transfers between user logic and devices. The virtual I/O module is the glue logic between user logic and the Shell; it abstracts the I/O specifications of the Shell (e.g., the number of I/O ports, supported device types, data width of stream paths) from user logic. The vFPIO scheduler is responsible for monitoring DMA requests issued by the MMUs and reordering them based on priorities assigned to user logic. Applications running on the host can configure I/O connectivities and priorities for user logic through custom APIs offered by the vFPIO library.

4.2 I/O Virtualization

We introduce the virtual I/O module that abstracts I/O devices from user logic. The virtual I/O module is designed to realize two key functionalities. First, it allows the user logic to have any number of stream I/O ports. Existing FPGA platforms directly expose device-specific I/O ports to the user logic region (vFPGA) [5, 34], which forces developers to carefully design I/O interfaces of their user logic so that they can properly connect to the given external I/O ports and consume/produce data streams. Second, it enables

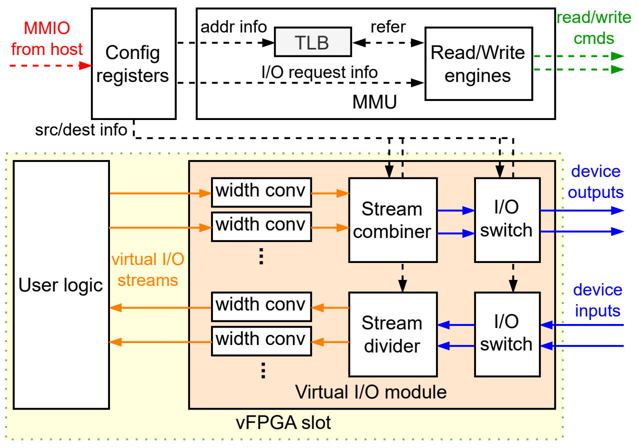

Figure 3: The virtual I/O module.

dynamically configuring the source and destination of virtual I/O ports. Modifying target I/O sources/destinations on existing FPGA platforms will also change the user logic, which forces users to re-synthesize their hardware design and reconfigure the FPGA. This process is time-consuming, and the engineering efforts are not negligible.

The virtual I/O module is a simple hardware module that enables dynamic I/O switching and data-flow control among user logic and I/O devices. The virtual I/O module fulfills the design goals above. First, the virtual I/O module liberates developers from the I/O constraints; the developers do not have to be aware of the number and specification (e.g., bus width) of the I/O ports exposed from the provided Shell. The virtual I/O module is synthesized as a part of the user logic and is configured in vFPGA slots. It enables configuring the virtual I/O to offer sufficient and proper I/O ports based on the user logic design. Second, the virtual I/O module has configurable I/O switches to avoid statically connecting the user logic to the dedicated I/O ports of devices. It allows user applications to change interconnects dynamically between user logic and devices.

Architecture. Figure 3 illustrates the virtual I/O module and a data flow between user logic and I/O devices. The virtual I/O module consists of three modules: width converters, stream combiner/divider, and switches. The width converters are tuned for each I/O port of user logic and convert them into the fixed port width supported by the Shell. The stream combiner is responsible for merging multiple streams from user logic to the same device. Meanwhile, the stream divider distributes a single input stream from the device into multiple output streams to user logic. We note that the width converters and stream combiner/divider are optional; they can be omitted if I/O ports of user logic follow the same I/O protocols supported by the Shell to suppress FPGA resource consumption. The switches inside the virtual I/O module route the I/O stream from DMA into corresponding virtual I/O ports and vice versa. Dynamic I/O switching driven by the virtual I/O module mitigates engineering efforts to modify/resynthesize user logic based on the source/destination I/O devices.

The virtual I/O module exposes an arbitrary number of *virtual I/O ports* to user logic, while the number of device-side I/O ports is fixed. The number of virtual I/O ports and their data widths are customized for each user logic and synthesized as a part of the vFPGA region. The connections between them can be dynamically configured by vFPIO file APIs (described in § 4.3). These APIs allow host applications to configure the switch integrated into the virtual I/O modules to establish the connections between virtual I/O ports and device-specific I/O ports. The MMU integrated into the Shell is not connected to the data stream buses but is responsible for issuing read/write requests to dedicated DMAs according to read/write requests from the host. The MMU exposes control paths to receive the requests from host applications through the vFPIO APIs.

4.3 vFPIO Library

The vFPIO library is designed to allow host applications to dynamically configure I/O connections between user logic and I/O devices without reconfiguring FPGA. To enhance programmability, vFPIO abstracts I/O devices as *files* like Linux device drivers [23] and offers POSIX-like file APIs (vFPIO APIs) to manage I/O-related operations. It decouples platform-dependent I/O logic from user logic designs, which makes user logic portable for different I/O devices. Although such a file-based I/O abstraction is similar to the memory virtualization mechanism offered by FSRF [36], vFPIO is the first work that adopts file I/O abstractions to various I/O devices.

vFPIO file APIs. Table 1 shows vFPIO file APIs that allow host applications to configure I/O interconnects during the execution. The host application first invokes vfpio_init() to obtain an available slot on FPGA and program the bitstream there. The API returns *vfpga_fd*, which indicates the instantiated user logic. vfpio_huge_alloc() allocates memory for storing the input or output data. vfpio_open() establishes an I/O connection between an I/O port of user logic, optionally specified by *port_id*, and the device specified by *device_type*. Once the devices for all the I/O ports are determined, the application can call vfpio_read() and vfpio_write() to perform data transfer. Those APIs invoke DMA transfer: write parameters (e.g., data size, memory address) to the control registers of the corresponding DMAs. vfpio_close() disconnects a specific port of user logic from the device indicated by *dev_fd*. vfpio_release() makes the obtained FPGA slot free.

Programming model. The vFPIO programming model follows industry-standard programming languages for heterogeneous devices like CUDA and OpenCL. As well as their programming models, the vFPIO application consists of kernel and host codes. Listing 1 shows a simple example of the kernel code written in Vitis HLS C++. We note that some vendor-provided platforms (e.g., Xilinx Vitis) require users to map kernel's I/O ports to devices using either pragma (`#pragma HLS_INTERFACE`) or HLS compiler options (`--connectivity.sp`) [58], while vFPIO avoids embedding such I/O-dependent information to the kernel

vFPIO APIs	**Description**
int vfpio_init(*bitstream*) int vfpio_release(*slot_id*)	Programs the bitstream to any free vFPGA slot. It returns *slot_id* if succeeded. Release the obtained vFPGA slot.
uint64_t* vfpio_huge_alloc(data_size)	Allocate memory for input and output data.
int vfpio_open(*device_type, rw_flag, slot_id*) int vfpio_open(*device_type, rw_flag, slot_id, port_id*) int vfpio_close(*dev_fd*)	Connects all I/O ports of user logic to the same device. Return a handle *dev_fd.* Connects an I/O port of user logic given by *port_id* to a device. Return *dev_fd.* Disconnects the device from the user logic.
size_t vfpio_read(*dev_fd, *address, size*) size_t vfpio_write(*dev_fd, *address, size*) int vfpio_sync(**dev_fds*)	Invokes data transfer from the device to the user logic. Invokes data transfer from the user logic to the device. Wait for completion of on-the-fly data transfers for *dev_fds.*

Table 1: The definition of vFPIO file APIs. The *address* is specified only when the target device supports memory-mapped I/O.

code and allows host code to change I/O connectivities using vFPIO file APIs without reconfiguring FPGA.

Listing 2 shows a simple example of host code written in C. The host code is responsible for initializing vFPGA slots with a bitstream of the kernel code and configuring I/O connections. This example configures the sha256 kernel shown in Listing 1 on FPGA. It invokes the execution by reading inputs from the specified input stream (FPGA memory and RDMA memory) and writing outputs to the output stream (FPGA memory and RDMA memory). In the vFPIO programming model, the host code configures user logic (kernel) to any *vFPGA slot*, representing a dynamic FPGA region (Line 2). It then maps the input/output ports of the kernel to any supported I/O devices that are abstracted as special files (Lines 11 and 21). Once I/O connections between kernel ports and devices are established, the host code can invoke data transfers among them (Lines 14-15 and 24-25). The second argument of vfpio_read() and vfpio_write(), *addresses*, can only be specified if the target device is byte-addressable memory. We assume that the underlying Shell provides a virtual address space of those memories for each vFPGA region as supported by the state-of-the-art Shells (Coyote [34], AmorphOS [32]). If asynchronous reads/writes are specified by vfpio_open() with `ASYNC_RD/WR` flags, the host code can call vfpio_sync() to wait for completion of data transfers (Lines 16 and 26).

4.4 vFPIO Scheduler

The vFPIO scheduler is designed to prevent I/O bus contentions caused by concurrent read/write requests from multiple accelerators, which fulfills the following design requirements. First, the scheduler should be device-agnostic: it needs to manage individual data streams transferred from/to these I/O devices because I/O contentions can occur at every I/O port of different devices. Second, the scheduler design should be as simplified as possible to be adapted to various FPGA platforms where different types/numbers of I/O devices and vFPGAs are supported.

To meet these requirements, vFPIO adopts a modular design with multiple scheduler instances. Each I/O device on the FPGA board is paired with two dedicated scheduler instances employing the same scheduling algorithms, one for read requests and one for write requests. This approach enables efficient and independent management of each I/O device. We note that the scheduler does not directly manipulate I/O streams but reorders I/O requests issued by MMUs before the dedicated I/O controllers or DMAs invoke actual data transfers. The scheduler only reorders requests across different vFPGAs but retains the order within each application to keep data integrity. This design reduces the complexity of each scheduler instance and ensures scalability; the Shell design can be flexibly extended in correspondence with the number of supported I/O devices and vFPGAs.

Architecture. Figure 4 illustrates the detailed design of the vFPIO schedulers. The vFPIO scheduler instance consists of two scheduler modules for read and write. Their architecture and applied scheduling logic are the same, while they differ only in the type of receiving requests. The schedulers receive priority control information from the host and read/write requests as input. The schedulers' output is connected to the DMA and the multiplex modules in the DMA input and output data stream. The multiplex module selects a single data stream from all vFPGA regions for the DMA, as the DMA can only handle one request at a time.

The scheduler consists of three components: a switch that selects the next request to execute, a completion FIFO to store signals for output, and a multiplex FIFO to store the multiplex control information. The FIFOs serve as buffer queues to store I/O requests waiting to be processed. Once the corresponding requests are processed, the FIFOs return the completion signals to the input DMA request interfaces. The request switch decides which vFPGA gets the highest priority for the subsequent requests based on priority information from the host. The priority for each vFPGA region is set by the host application, and it is applied to all I/O requests initiated/targeted from that vFPGA until the host application updates the priority. The switch can change the applied policy between the Round-Robin and the First-in-First-out (FIFO) based on the priority setup of all vFPGA regions.

When all vFPGAs have the same priority, the scheduler adopts the Round-Robin policy so that each vFPGA gets equal timeshare in circular order. If vFPGAs have different priorities, the scheduler chooses the FIFO policy; vFPGAs are served based on priority from highest to lowest, and user logic on a vFPGA region continues the execution until it finishes or is preempted by a higher priority one. The FIFO policy is a preemptible one that can guarantee faster execution for high-priority workloads.

```
void sha256(
hls::stream<ap_axiu<AXI_DATA_BITS, 0, PID_BITS, 0> >& in,
hls::stream<ap_axiu<AXI_DATA_BITS, 0, PID_BITS, 0> >& out
) {
    hls::stream
      <net_axis<AXI_DATA_BITS>> inStream("input_stream");
    hls::stream<
      net_axis<AXI_DATA_BITS>> outStream("output_stream");

#pragma HLS dataflow
    read_input(in, inStream);
    sha256_compute(inStream, outStream);
    write_result(out, outStream);
}
```

Listing 1: An example of kernel code to compute SHA256 hashes over input data written in Vitis HLS C/C++. The kernel contains generic I/O interfaces that can be switched across multiple I/O types at runtime.

```
void exec_sha256(void* bitstream) {
    int vfpga = vfpio_init(bitstream);
    int dev_fd;

    uint64_t
      *fMem = (uint64_t *)vfpio_huge_alloc(DATA_SIZE);

    IODev dev1 = IODevs::RDMA_MEM;
    IODev dev2 = IODevs::FPGA_MEM;

    /* Configure kernel to use RDMA memory */
    dev_fd = vfpio_open(dev1, ASYNC_RDWR, vfpga);

    /* Transfer data and run kernel code */
    vfpio_read(dev_fd, fMem, DATA_SIZE);
    vfpio_write(dev_fd, fMem, DATA_SIZE);
    vfpio_sync(dev_fd);
    vfpio_close(dev_fd);
    /****************************************/

    /* Configure kernel to use FPGA memory */
    dev_fd = vfpio_open(dev2, ASYNC_RDWR, vfpga);

    /* Transfer data and run kernel code */
    vfpio_read(dev_fd, fMem, DATA_SIZE);
    vfpio_write(dev_fd, fMem, DATA_SIZE);
    vfpio_sync(dev_fd);
    vfpio_close(dev_fd);
    /****************************************/
}
```

Listing 2: An example of host code using vFPIO file APIs.

5 Implementation

We implement the vFPIO framework as an extension of Coyote [34] on AMD Xilinx Alveo U280 cards. We have considered other FPGA OSes, AmorphOS [32] and FSRF [36], but the former is not open-sourced, and the latter only supports memory I/Os. Moreover, AmorphOS and FSRF target Amazon's cloud FPGA instances [3] and do not support commercially available FPGA cards.

5.1 vFPIO Shell

We extend the Coyote Shell to apply our I/O virtualization and scheduling mechanisms. The Coyote Shell on U280 supports

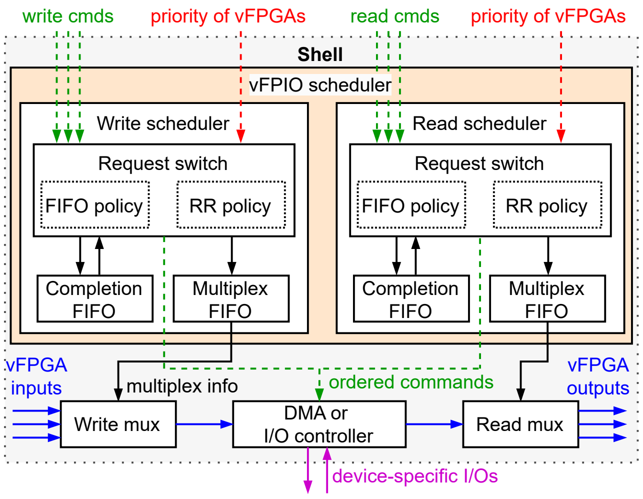

Figure 4: The vFPIO scheduler module.

four external I/Os: PCIe, DRAM, HBM, and 100 Gbps network. It adopts the AXI stream interface [19] for data paths between user logic and dedicated I/O controllers. We integrate multiple vFPIO scheduler modules into the Coyote Shell to individually manage these I/O interfaces. Coyote supports RDMA and TCP/IP stacks for network interfaces [49, 50]. Since we cannot enable both stacks simultaneously, we adopt the RDMA stack for the vFPIO Shell in this paper. We also extend the control interface between host applications and the vFPIO Shell by adding memory-mapped registers to configure the I/O switch of the virtual I/O modules and set up the priority. Like Coyote, vFPIO does not fully support direct data transfers between different I/O devices.

vFPIO Scheduler. We implement the vFPIO scheduler by extending the TLB arbiter module of the Coyote Shell. Because the read/write command protocols for the dedicated I/O controllers and DMAs are different, we customize each scheduler module to their target I/Os and integrate them into the Shell. The I/O transaction requests issued by host applications are routed to the corresponding scheduler module.

Our implementation features a new preemptible scheduling policy in addition to the Round-Robin policy, which is supported by the TLB arbiter for all variants of scheduler modules that always serve transactions with the highest priority. The request switch chooses the scheduling policy based on priority information from the host and selects the next request to execute. The priority list for all vFPGA regions can be updated dynamically using vFPIO APIs. The changes are immediately applied from the next clock cycle.

5.2 Virtual I/O Module

We implement the virtual I/O module as an HDL wrapper written in SystemVerilog. The virtual I/O module uses the AXI stream protocol for data paths as well as the vFPIO Shell. The virtual I/O module is synthesized with user logic written in HDL and Vitis HLS C++ [14]. The module has application-specific parameters, such as the number of virtual

Server (Host and FPGA)	
Host CPU	AMD(R) EPYC 7413 24 cores @2.65 GHz
OS	NixOS 23.0, Linux 6.6
Memory	256 GiB DRAM
FPGA	Xilinx Alveo U280
HBM2 Capacity	8 GiB
HBM2 Bandwidth	460 GiB/s
PCI Express	Gen4x8 with CCIX
Network Interfaces	2x QSFP28 (100GbE)

Table 2: Server and FPGA specification.

Name	Description
aes	AES-256 crypto
sha256	SHA-256 hashing
md5	MD5 hashing
nw	Needleman-Wunsch
matmul	Matric multiplication
sha3	Keccak-256 hashing
rng	Random sequence
gzip	gzip compression
perf-(IO)	Data streaming using (IO)

Table 3: FPGA benchmarks used in the evaluations.

I/O ports and port widths, which must be configured for each user logic before running synthesis. Meanwhile, the I/O switch in the module can be dynamically configured through the memory-mapped control registers. The virtual I/O module supports switching between host (PCIe) DRAM, FPGA DRAM, HBM, and RDMA stack on U280.

5.3 vFPIO Library

We implement vFPIO APIs, listed in Table 1, as a C++ library. All the APIs manipulate memory-mapped control registers of the Shell through the device driver and PCI subsystem. The vfpio_init(), vfpio_release(), vfpio_open(), and vfpio_close() APIs dynamically configure the selected vFPGA slot. The vfpio_init() invokes FPGA reconfiguration with the selected bitstream through the ICAP [4] interface. It also opens a memory-mapped region between the host and the FPGA for I/O data communication. The obtained slot is released by vfpio_release(). The vfpio_open() transmits the I/O device type to the virtual I/O module by setting the I/O type to the module's control registers. As a result, the switch is configured to connect the specified user logic's I/O port to the device. The vfpio_close() performs the reverse operation, clearing the configuration of the I/O switch and notifying the corresponding scheduler.

The vfpio_read() and vfpio_write() APIs invoke actual data transfers via data paths. They initiate read and write operations of the selected I/O device by writing information, such as the data size and source/destination memory addresses, to the control registers. The read/write engines of the Coyote MMU react to the register values and generate read/write commands of the corresponding controllers/DMAs. The vfpio_sync() is realized by a CPU thread polling the status register of the corresponding I/O controller or DMA.

5.4 vFPIO Portability

The virtual I/O concept of vFPIO can be extended to other FPGA frameworks, such as AmorphOS [32] and Optimus [38], by implementing the vFPIO scheduler and the virtual I/O modules. These components can work on other FPGA devices since they do not have special hardware resource requirements. The virtual I/O components will not significantly affect the performance of other FPGA OSes due to their design simplicity.

6 Evaluation

We evaluate the vFPIO framework from the following dimensions: performance (§ 6.1), programmability (§ 6.2), portability (§ 6.3), scheduler (§ 6.4), and resource overheads (§ 6.5).

Experimental setup. We perform the experiments on two servers configured as shown in Table 2. The two U280 FPGAs on different servers are connected through a 100Gbps network cable for the RDMA setup. We configure the vFPIO Shell with up to three virtual FPGA regions to enable the parallel execution of multiple applications.

Benchmarks. Table 3 shows the applications we use for evaluations. The perf-(IO) has two variants, perf-host and perf-fpga, one using host DRAM and the other using FPGA HBM as the I/O devices used for data transactions. We have considered more complex applications as benchmarks; however, they have few advantages over existing ones regarding I/O throughput on FPGA. We believe the current set of benchmarks represents a wide group of applications and can sufficiently demonstrate vFPIO's properties.

6.1 Performance

We evaluate the end-to-end performance to show the overheads introduced by vFPIO.

Methodology. We measure the I/O throughput of benchmark applications to evaluate vFPIO's performance overheads. Since no storage driver exists that can execute directly on an FPGA, we emulate a storage device using the FPGA's local memory, i.e., the HBM. Through this setup, we emulate a near-data computing environment wherein applications can offload specific logic to be executed close to the location of the input data in the HBM. We also consider an RDMA setup, where input data is fetched from remote storage, i.e., host DRAM on a remote node, and the application invokes RDMA reads and writes to obtain the data over the network. The application's output is written to the local host DRAM. We evaluate vFPIO against two baselines: a host-only baseline (`Host`) that executes the entire application on a host CPU and Coyote (`Coyote`) that offloads a computational task to user logic running on the original Coyote Shell. In the `Host` case, the host application reads input data from FPGA HBM and executes the computation. We include CPU-only cases to highlight the performance advantages of FPGA compared with CPU-only setups. Whereas, in the cases of vFPIO and

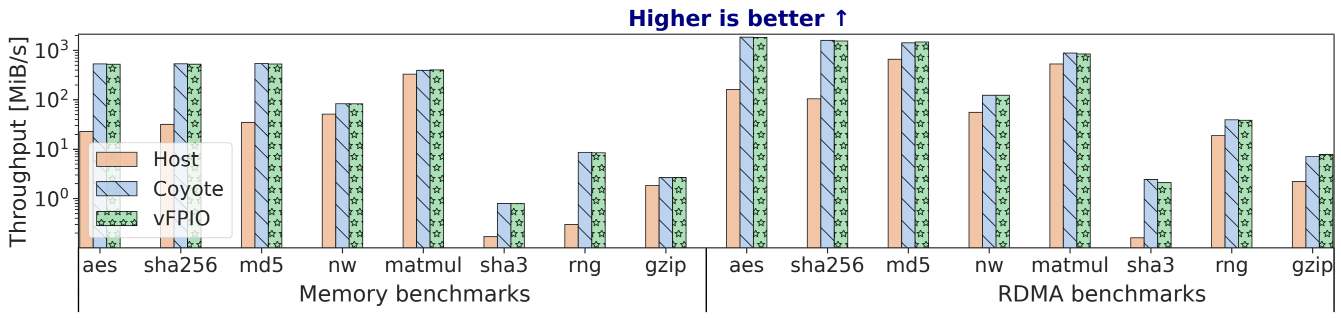

Figure 5: vFPIO end-to-end overhead of applications. Memory benchmarks show the performance using FPGA HBM, and RDMA benchmarks show the one using the FPGA RDMA stack. Throughput is measured with the benchmarks running alone on the host, Coyote, and vFPIO.

Coyote, the computational logic offloaded to FPGA directly reads the input data and transfers the results to the host memory. For the RDMA case, we use two nodes for the experiments: one server and one client. The server node sends the input data from the host DRAM to the client node using RDMA. The transferred data is either stored in host memory on the client node for the host-only computation (`Host`) or directly executed by the user logic on FPGA (vFPIO and Coyote). We use one vFPGA region to instantiate the user logic for vFPIO and Coyote. The scheduler uses the default Round-Robin scheduling policy for both setups. We repeat the execution of each application ten times and report the mean value.

Results. Figure 5 shows the average throughput of the applications on a log scale. The average percentage of standard variance to mean for all benchmarks is 18.4%. Overall, we see that both FPGA solutions (vFPIO and Coyote) achieve higher performance for both memory and RDMA I/O setups than the host-only baseline. For memory I/O applications, vFPIO achieves, on average, 11.5 × faster than the host baseline, and the performance difference between Coyote is only 0.7% on average for all applications. For applications using RDMA, the throughput on vFPIO is, on average, 6.3 × faster than the host baseline with a 1.1% of performance difference compared with Coyote. Note that with RDMA, applications have better throughput than using FPGA HBM. This performance difference is because the RDMA stack uses a streaming interface and sends data directly to the host DRAM without using FPGA HBM. This setup avoids the latency of writing and reading from FPGA HBM for I/O. These results indicate that FPGA task offloading significantly improves the performance of I/O-intensive workloads compared to the CPU-only solution for both the vFPIO and Coyote setups. In addition, the performance difference between vFPIO and Coyote for both I/O devices is negligible, showing that our virtual I/O extension does not affect the application's I/O throughput.

Summary. vFPIO outperforms the CPU-only host baseline for all I/O intensive benchmarks (11.5 × for HBM and 6.3 × for RDMA) and incurs negligible overhead compared with Coyote (0.7% for HBM and 1.1% for RDMA).

	SLoC		CC	
Applications	**Coyote**	**vFPIO(%)**	**Coyote**	**vFPIO(%)**
Host code	141	58 (58.9)	21	5 (76.2)
aes	892	857 (3.9)	64	64 (0)
sha256	599	441 (26.4)	67	61 (9)
md5	700	592 (15.4)	55	51 (7.3)
nw	870	859 (1.3)	95	95 (0)
matmul	301	290 (3.7)	45	45 (0)
sha3	456	445 (2.4)	75	75 (0)
rng	455	434 (4.6)	53	53 (0)
gzip	240	209 (12.9)	0	0 (0)

Table 4: Source Lines of Code (SLoC) and Cyclomatic Complexity (CC) comparison between Coyote and vFPIO for the host (CPU) application code and user logic. The table also shows the percentage reduction in SLOC and CC for vFPIO compared to Coyote.

6.2 Programmability

We evaluate the programmability benefits of vFPIO for both host code and FPGA user logic.

Methodology. We evaluate the programmability benefits along two dimensions: source lines of code (SLOC) and cyclomatic complexity (CC). The two metrics measure how many I/O control/interface codes developers need to write, thus indicating the required programming effort. We employ *scc* [44] to obtain these metrics for FPGA user logic and host application code that controls and communicates with the user logic. We evaluate these metrics for both Coyote and vFPIO. The host code is written in C++, and the user logic is written in SystemVerilog and VHDL.

Results. Table 4 shows the SLOC and CC metrics of our benchmarks for both Coyote and vFPIO. For the C++ host code, we only show SLOC and CC for a common template used for all applications. The result highlights that vFPIO APIs achieve a 58.9% reduction in SLOC and a 76.2% reduction in CC in host code compared to Coyote. In Coyote, switching between various I/O devices for the same application requires special reconfiguration code written in the host code on a per I/O device basis. vFPIO avoids this additional effort because the virtual I/O module decouples device-specific I/O interfaces from user logic, and the vFPIO APIs provide easy-to-use I/O abstractions for switching between multiple I/O devices. Similarly, for the FPGA user logic, we show a maximum

	Throughput				Reconfiguration time		
	vFPIO [MiB/s]				Coyote [ms]		vFPIO [μs]
App	HBM	(%)	Host	(%)	HBM	Host	
aes	525.2	(98.3)	1895.6	(105.9)	21.0	21.1	1.3
sha256	527.4	(98.3)	1865.5	(96.2)	20.2	19.4	1.3
md5	531.9	(98.3)	1889.1	(99.6)	18.8	17.6	1.3
nw	82.92	(98.8)	304.9	(102.1)	29.2	29.3	1.3
matmul	404.4	(102.1)	1418.6	(97.2)	23.9	23.7	1.3
sha3	0.79	(99.2)	2.85	(101.0)	19.4	17.7	1.3
rng	8.43	(97.0)	30.69	(99.2)	17.1	17.0	1.3
gzip	2.66	(100.8)	9.81	(103.2)	23.6	23.9	1.3

Table 5: Throughput and I/O reconfiguration time using FPGA HBM and host memory on vFPIO and Coyote. The relative (%) compares with the throughput on Coyote.

reduction of 26.4% for SLOC and 9% for CC. This is because in vFPIO, due to the decoupling between the computation logic and I/O logic, the user logic has a generic I/O interface that can be switched between I/O devices at runtime. Whereas, in Coyote, the user logic has to be rewritten to interface with specific I/O devices. These results indicate that vFPIO reduces the codes developers need to write for both host applications and FPGA user logic and reduces program complexity, thus improving the programmability of FPGA.

Summary. FPGA I/O configuration driven by the vFPIO APIs simplifies the design of both the host code (reduce 58.9% SLOC, 76.2% CC) and the user logic (reduce 26.4% SLOC, 9% CC) by decoupling the computation logic from device-specific I/O interfaces.

6.3 Portability

We evaluate vFPIO's portability advantages against Coyote.

Methodology. To evaluate the portability, we compare the I/O switching overhead when an application changes user logic's I/O connection from one device to another for both vFPIO and Coyote. First, we prepare two partial bitstreams for each application for Coyote, one using FPGA HBM and one using host memory as an I/O device, and then measure the time required to reconfigure the vFPGA region with each bitstream. For vFPIO, we prepare only one partial bitstream for each benchmark, where the user logic and virtual I/O module are configured to dynamically switch I/O interfaces between HBM and host memory. We measure the time required to switch between the FPGA's HBM and host memory using vFPIO's I/O switch. Additionally, to show the overheads of vFPIO's I/O switch, we compare the I/O throughput in MiB/s for both Coyote and vFPIO.

Results. Table 5 shows the application throughput and the reconfiguration time for both vFPIO and Coyote. With dynamic I/O switching enabled, vFPIO achieves nearly the same (at most 3.8% less) throughput for all benchmark applications compared to Coyote with both FPGA HBM and host memory. The result shows that vFPIO's I/O switching mechanism does not affect applications' performance. For the reconfiguration time, vFPIO achieves 1.3 μs for all benchmark applications, while Coyote requires, on average, 22.7 ms for HBM and 22.4 ms for host memory. We note that vFPIO's I/O reconfiguration time is the same for all benchmark applications because the switching mechanism does not depend on applications. This significant improvement compared to Coyote is because vFPIO only requires changing a control register value on the FPGA, after which the virtual I/O module switches the connection between user logic and I/O devices without reconfiguration. On the other hand, Coyote has to reconfigure the entire vFPGA region to switch the user logic between various I/O devices. The reconfiguration time on Coyote depends on the application's partial bitstream size (on average, 13.8 MiB in our case), and it is much more time-consuming than simply writing values to memory-mapped control registers.

Summary. The vFPIO framework allows user logic to dynamically switch I/O connections among different I/O devices with less reconfiguration time (1.3 μs) than Coyote (22.7 ms for HBM, 22.4 ms for Host). It does have negligible throughput penalties (3.8% at worst).

6.4 Scheduler

We evaluate the scheduler's effectiveness for performance isolation across multiple applications and associated overheads.

6.4.1 Performance Isolation

We first evaluate the ability of the scheduler to provide performance isolation across concurrent applications.

Methodology. To demonstrate the effectiveness of the priority-based I/O scheduling, we configure multiple vFPGAs with different I/O scheduling priorities and observe the execution time when running all application instances concurrently. We configure the three instances of perf-host, which transfer 1,000 chunks of 16 kB data between host and FPGA via PCIe, and measure their execution times when run concurrently. We configure one instance with high I/O scheduling priority (HP) and the others with lower priorities (LP-1 and LP-2). For comparison, we run the same experiment with the original Coyote Shell, where the round-robin policy (RR) is applied, and measure the average execution time of the three instances.

Results. Figure 6 (a) shows the execution times of three instances of the application in terms of the number of cycles for high-priority (HP) and low-priority (LP-1 and LP-2) instances. The figure also shows the average execution times of three application instances for round-robin (RR). We observe that the execution of the HP instance is 1.8 $\times$ faster than the LP-1 and LP-2 instances and 1.7 $\times$ faster than the RR instance because the vFPIO scheduler always prioritizes I/O requests issued from the HP instances. The scheduler suspends I/O requests from the two LP instances until the HP instance is finished and resumes them concurrently in a round-robin fashion. The result indicates that the vFPIO scheduler preserves the I/O throughput for high-priority tasks. We also observe that the execution time of the two LP instances is longer than the RR

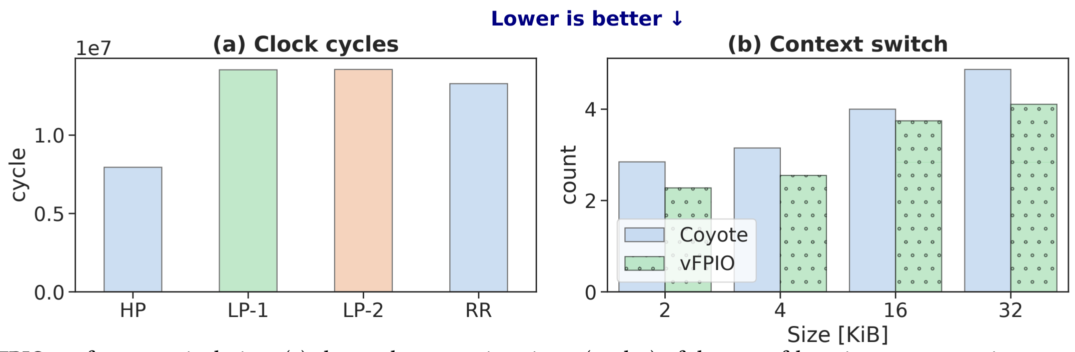

Figure 6: vFPIO performance isolation. (a) shows the execution times (cycles) of three perf-host instances running concurrently. HP, LP-1, and LP-2 are in the vFPIO case where one instance has high priority (HP), and the others have low priority (LP-1, LP-2), and RR is in the Coyote case where all three instances run in a round-robin fashion. (b) shows the number of context switches in both scenarios.

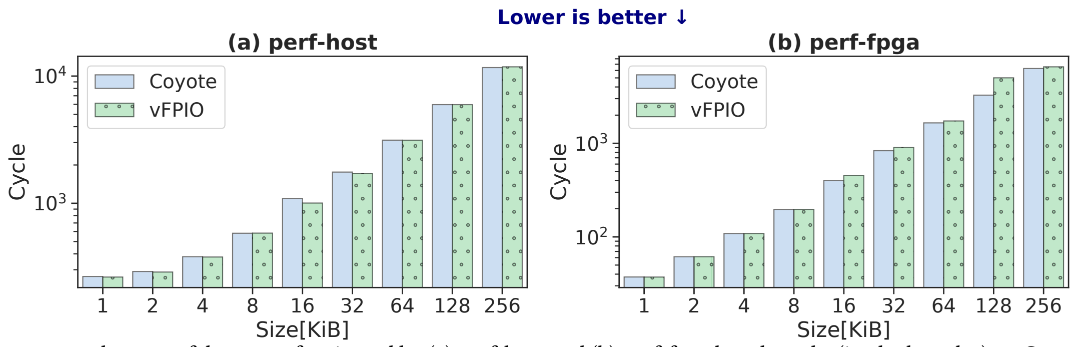

Figure 7: The average latency of data transfers issued by (a) perf-host and (b) perf-fpga benchmarks (in clock cycles) on Coyote and vFPIO.

instance, while the performance difference is only 6.3%. One factor is that the vFPIO scheduler mitigates the number of context switches between I/O requests from different instances, as shown in Figure 6 (b). Fewer context switches mean memory accesses are more continuous and fewer page faults happen, mitigating the performance penalties for low-priority tasks.

6.4.2 Scheduling Overheads

We next evaluate the scheduler's I/O performance overheads.

Methodology. To break down the overhead incurred by the vFPIO scheduler, we configure two applications (perf-host, perf-fpga) that invoke data transfers on the vFPIO and Coyote and measure the average latency of data transfers on the different platforms. We measure the average latency of 1,000 data transfers while changing data sizes from 1 kB to 256 kB.

Results. Figure 7 shows the average latency measured by the two microbenchmarks. For both benchmarks, the data transfer latency on the vFPIO is equivalent to Coyote. These results indicate that the vFPIO scheduler overhead does not induce additional performance overheads compared to Coyote's I/O scheduler, which supports the round-robin policy only.

Summary. The vFPIO scheduler arbitrates concurrent I/O requests from multiple vFPGAs to maximize the throughput of high-priority tasks (1.7 × faster than round-robin among three vFPGAs).

6.5 Resource Overheads

We evaluate the FPGA resource overheads induced by vFPIO.

Name	**LUT** (%)		**FF** (%)		**BRAM** (%)	
U280	1303680	(100)	2607360	(100)	2016	(100)
Coyote	338727	(26.0)	687011	(26.3)	408	(20.2)
vFPIO	344506	(26.4)	700906	(26.9)	408.5	(20.3)
RDMA	172958	(13.3)	427219	(16.4)	213	(10.6)
PCIe DMA	59089	(4.5)	58114	(2.2)	80	(4.0)
Local DMA	13691	(1.1)	11146	(0.4)	0	(0.0)
MMU	8102	(0.6)	13773	(0.5)	20	(1.0)
Scheduler	270	(0.0)	513	(0.0)	0	(0.0)
Virtual IO	3	(0.0)	3090	(0.1)	0	(0.0)

Table 6: vFPIO's resource usage. The relative (%) compares with the available resources on the U280 FPGA. Coyote and vFPIO represent the entire Shell.

Methodology. We break down the resource usage of the vFPIO Shell using Vivado IDE, which includes lookup tables (LUTs), flip-flops (FFs), and block RAM (BRAM). We report the entire usage of the Shell and individual usages of various primary hardware components. We also report the resource utilization of the Coyote Shell to verify the resource overhead induced by our hardware extension. Both vFPIO and Coyote Shells have one vFPGA slot, and the vFPIO connects to the host (PCIe) interface, local memory (DRAM), and two network ports via RDMA.

Results. Table 6 shows the resource usage of hardware components. Overall, the vFPIO Shell consumes 26.4% of LUTs, 26.9% of FFs, and 20.3% of BRAMs in the U280 FPGA. We observe that 67.4% of LUTs and 69.1% of FFs are consumed by Coyote's RDMA stack and the vendor-provided IP core (PCIe DMA, i.e., XDMA). Compared to the Coyote Shell, vFPIO induces only 0.4% and 0.6% resource overheads of LUTs and FFs, respectively. The results also indicate that the scheduler and virtual I/O modules use fewer resources than the dedicated DMAs. We note that these modules consume more resources as the number of vFPGAs and supported I/O devices increases, while individual modules are small enough to scale up.

Summary. vFPIO incurs negligible resource overheads (0.4% LUTs, 0.6% FFs) compared to the state-of-the-art Shell. The individual scheduler and virtual IO modules are small enough, even considering the scalability.

7 Related Work

FPGA I/O virtualization. FPGA virtualization studies have applied OS primitives to hardware tasks to enable multi-tasking [21, 25, 28, 54, 59], memory virtualization [18, 22, 60], and host-FPGA communication [31, 39], while I/O virtualization is not well explored. FPGA OSes [32, 34, 36, 53, 57] focus on virtualizing memory devices and do not consider the other device types. ViTAL [61–63] splits reconfigurable regions of FPGA devices into segments and constructs a single, virtualized slot over them. Unlike vFPIO, it does not virtualize external I/Os from the FPGA. AmorphOS [32] and FOS [57] provide FPGA resource sharing and isolation among distrusted processes. They statically embed a wrapper module to user logic to mediate external I/O accesses, while dynamic I/O configuration is not supported. In contrast to these studies, vFPIO offers a unified way to abstract various types of external I/O devices and enables dynamic switching of I/O connections.

FPGA programmability and portability. Prior studies provide abstracted programming models to ease FPGA application development and performance tuning. A compiler-based approach is commonly used for kernel code programming. Merlin compiler [15] provides an OpenMP-like programming model and pragmas for parallelization and pipelining. COMBA [65] uses a model-based analysis framework to generate high-performance pragma configuration. Cascade [45] and Synergy [37] allow kernel code to describe I/O data flows on FPGA at a language level. These approaches embed I/O information into kernel code, which harms the code portability. Unlike them, vFPIO decouples device-specific I/O information from kernel code, which gains portability.

For host code programming, Catapult [42] comes with a standard driver and user-level interface for communication between the host application and FPGA. VirtualRC [33] allows developers to design user logic with configurable interfaces that run on the provided virtual FPGA platform. BORPH [53] and FSRF [36] propose a filesystem-based FPGA memory abstraction. They allow host code to operate memory reads/writes on FPGA using file APIs. The vFPIO framework further extends the file-based abstraction to cover diverse I/O interfaces supported by modern FPGA devices.

FPGA I/O scheduling. FPGA task scheduling mainly focuses on improving FPGA resource usage with time-sharing among multiple tasks [24, 25, 34, 59] and accelerator reuse that avoids frequent partial reconfiguration [57]. AmorphOS [32] focuses more on spatial sharing and mitigates the resource overhead caused by fragmentation due to partial reconfiguration. The vFPIO scheduler works independently from these task schedulers and potentially cooperates with them to achieve better resource utilization and higher I/O throughput.

Regarding I/O scheduling, most existing studies focus on network packet scheduling [41, 43, 52]. FlowBlaze [41] presents an open abstraction for using stateful packet processing functions in programmable NICs. Sivaraman et al. [52] propose a programmable packet scheduler design that applies different scheduling algorithms without hardware redesign. On the other hand, Coyote [34] arbitrates read/write requests for every I/O device, but it simply provides a fair share for all the accelerators configured in the vFPGA slots. The vFPIO scheduler is superior to these approaches because it supports a preemptible scheduling mechanism that can be applied to any type of I/O transaction.

8 Conclusion

This paper introduces vFPIO, an FPGA-based I/O acceleration framework with the following tangible contributions:

1. **I/O virtualization for user logic.** We propose a new notion of I/O virtualization for FPGA accelerators that improves design portability. The proposed virtual I/O module abstracts I/O interfaces of external devices and decouples the device-specific specifications from user logic design.
2. **File abstraction for configurable I/Os.** We present POSIX-like file I/O APIs that dynamically configure I/O directions of user logic. They allow developers to easily reuse the same user logic to manage different I/O devices and data transfers without specific hardware knowledge.
3. **Device-agnostic I/O scheduling.** We present an I/O transaction scheduling mechanism that arbitrates multiple read/write requests from concurrent applications to prevent throughput degradation due to bus congestion.

We implement the vFPIO framework based on Coyote [34] for AMD Xilinx FPGA cards equipped with various I/O interfaces. vFPIO enables FPGA workloads to manage various I/O devices without changing the hardware design while introducing slight performance penalties.

Software artifact

We release vFPIO as an open-source project [13].

Acknowledgments

This work was partially supported by an ERC Starting Grant (ID: 101077577). The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany in the programme of "Souverän. Digital. Vernetzt.". Joint project 6G-life, project identification number: 16KISK002.

A Artifact Appendix

A.1 Abstract

This artifact contains the implementation and scripts to reproduce the experiments and figures from the USENIX ATC 2024 paper – "vFPIO: A Virtual I/O Abstraction for FPGA-accelerated I/O Devices" by J. Chen, H. Unnibhavi, A. Koshiba, P. Bhatotia. vFPIO is a hardware-software co-design framework that offers improved programmability and portability for FPGA I/O accelerators while ensuring the I/O throughput for high-priority tasks during resource contention.

A.2 Scope

The artifact is used to reproduce all the experimental results shown in the paper. It contains both hardware and software source code needed to compile the FPGA Shell and host applications and scripts to run the evaluation. We provide pre-compiled FPGA bitstreams of the Shell and benchmarks to save development time.

A.3 Contents

The artifact is published in our GitHub repository and contains the following items:

- `bitstreams/`: the pre-compiled FPGA bitstreams.
- `driver/`: source code for the Linux kernel driver.
- `hw/`: source code for the vFPIO Shell and user logic of benchmark applications.
- `sw/`: source code for host applications of benchmark applications.

The root path also contains all the scripts and files for reproducing the evaluation results.

A.4 Hosting

All the project source code, including the instructions for evaluating and building the software, is available in the following git repository: https://github.com/TUM-DSE/vFPIO. Please use the `vfpio` branch to reproduce the results.

A.5 Requirements

We require the following software and hardware configurations to reproduce our experimental results.

A.5.1 Hardware Dependencies

- Two machines with AMD EPYC 7413 CPU connected to public network.
- Two Xilinx (AMD) Alveo U280 FPGA cards.
- The two FPGAs should be directly connected using a QSFP28 network cable using network port 0 on the FPGA card.

A.5.2 Software Dependencies

- Operating system: NixOS 23.11 with Linux kernel 6.8.12.
- Nix: we use the Nix package manager to download all build dependencies for reproducibility. We use nix-shell to provide a consistent runtime environment.
- Python 3.11 or newer for the script that reproduces the evaluation.
- Vivado v2022.1 to compile the FPGA bitstreams and program the FPGA.

A.6 Methodology

The evaluation script `reproduce.py` reproduces the following results shown in the paper:

- Figure 5 (§ 6.1): throughput comparison of the three setups using memory benchmarks and RDMA benchmarks to show the performance overhead of vFPIO.
- Table 4 (§ 6.2): Source Lines of Code (SLoC) and Cyclomatic Complexity (CC) comparison between vFPIO and Coyote for the host (CPU) application code and user logic to show the improvement in programmability of vFPIO.
- Table 5 (§ 6.3): throughput and I/O reconfiguration time comparison of vFPIO and Coyote using FPGA HBM and host memory as I/O devices to show the fast reconfiguration of vFPIO.
- Figure 6 (§ 6.4.1): micro-benchmarks to show the effectiveness of performance isolation of vFPIO.
- Figure 7 (§ 6.4.2): micro-benchmarks to show the overhead of vFPIO scheduler.
- Table 6 (§ 6.5): hardware resource usage of vFPIO.

To run `reproduce.py`, one must prepare the Linux kernel driver, host application binaries, and FPGA bitstreams. The FPGA bitstreams are already provided in the GitHub repo to avoid the long compilation time.

To build the driver, run the following commands:

```
$ nix-shell vfpio.nix
$ cd driver
$ make
$ exit
```

The driver installation is handled by the evaluation script. However, to use the RDMA stack, the user needs to specify the MAC addresses of the FPGA cards. These values can be modified in the `program_fpga.sh` file (`mac_addr_q0`).

To build the host applications, run the following commands:

```
$ nix-shell vfpio.nix
# in the project repo root
$ bash compile_sw.sh
```

A.6.1 Performance (§ 6.1)

First, run the following commands to measure the benchmark throughput for different setups:

```
# requires nix-shell vfpio.nix
$ python3 reproduce.py -r -e Exp_6_1_host_list
$ python3
 ↪  reproduce.py -r -e Exp_6_1_coyote_list
$ python3
 ↪  reproduce.py -r -e Exp_6_1_vfpio_list
$ python3 reproduce.py
 ↪  -r -e Exp_6_1_host_rdma_list -s ip_address
```

```
$ python3 reproduce.py -r
 ↪ -e Exp_6_1_coyote_rdma_list -s ip_address
$ python3 reproduce.py
 ↪ -r -e Exp_6_1_vfpio_rdma_list -s ip_address
```

`ip_address` refers to the public IP address of the machine running the scripts.

To generate Figure 5, run the following commands:

```
$ python3 write_csv.py -e 6_1
$ python3 plot_e2e.py
```

A.6.2 Programmability (§ 6.2)

Run the following commands to generate a CSV file (`complexity.csv`) that contains the data to fill Table 4:

```
# requires nix-shell vfpio.nix
# in the project repo root
$ bash
 ↪ ./measure_complexity.sh > results_6_2.csv
$ python3 write_csv.py -e 6_2
```

A.6.3 Portability (§ 6.3)

Note that the throughput data in Table 5 uses the result of A.6.1. Run the following commands to measure and obtain the reconfiguration time for vFPIO and Coyote:

```
# requires nix-shell vfpio.nix
# in the project repo root
$ python3
 ↪ reproduce.py -r -e Exp_6_3_pr_host_list
$ python3
 ↪ reproduce.py -r -e Exp_6_3_pr_hbm_list
$ python3 reproduce.py
 ↪ -r -e Exp_6_3_host_coyote_list
$ python3
 ↪ reproduce.py -r -e Exp_6_3_host_vfpio_list
$ python3
 ↪ reproduce.py -r -e Exp_6_3_vfpio_list
```

Next, run the following command to generate a CSV file (`reconfig.csv`) that contains the data to fill Table 5.

```
python3 write_csv.py -e 6_3
```

A.6.4 Scheduler (§ 6.4)

First, run the following commands to obtain data for Figure 6 and 7:

```
# requires nix-shell vfpio.nix
# in the project repo root
$ python3
 ↪ reproduce.py -r -e Exp_6_4_1_cycle_list
$ python3
 ↪ reproduce.py -r -e Exp_6_4_1_cntx_list
$ python3
 ↪ reproduce.py -r -e Exp_6_4_2_host_list
$ python3
 ↪ reproduce.py -r -e Exp_6_4_2_fpga_list
```

To generate Figure 6, run the following commands:

```
$ python3 write_csv.py -e 6_4_cycle
$ python3 write_csv.py -e 6_4_cntx
$ python3 plot_iso.py
```

To generate Figure 7, run the following commands:

```
$ python3 write_csv.py -e 6_4_host
$ python3 write_csv.py -e 6_4_fpga
$ python3 plot_overhead.py
```

A.6.5 Resource Overheads (§ 6.5)

We provide the pre-generated resource utilization report files (`util_coyote.csv`, `util_vfpio.csv`) to save compilation time. These reports are generated by Vivado IDE. Run the following commands to extract the resource utilization of each component:

```
# requires nix-shell vfpio.nix
# in the project repo root
$ python3
 ↪ reproduce.py -e Exp_6_5_resource_util
```

References

[1] Alveo U250 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u250.html. Accessed: June 2024.

[2] Alveo U25N SmartNIC. https://www.xilinx.com/products/boards-and-kits/alveo/u25n.html. Accessed: June 2024.

[3] Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1. Accessed: June 2024.

[4] AMD: ICAP Interface. https://docs.xilinx.com/r/en-US/pg036_sem/ICAP-Interface. Accessed: June 2024.

[5] AMD OpenNIC Project. https://github.com/Xilinx/open-nic. Accessed: June 2024.

[6] Compute Express Link ™: The Breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org/. Accessed: June 2024.

[7] Intel ®Infrastructure Processing Unit (Intel®IPU) Platform F2000X-PL. https://www.intel.com/content/www/us/en/products/details/network-io/ipu/f2000x-pl-platform.html. Accessed: June 2024.

[8] Intel® FPGA SmartNIC. https://www.intel.com/content/www/us/en/products/details/fpga/platforms/smartnic.html. Accessed: June 2024.

[9] NVMe Specifications Overview. https://nvmexpress.org/specifications/. Accessed: June 2024.

[10] P2P Simple Example. https://xilinx.github.io/Vitis_Accel_Examples/2023.1/html/p2p_simple.html. Accessed: June 2024.

[11] PCI Express 6.0 Specification. https://pcisig.com/pci-express-6.0-specification. Accessed: June 2024.

[12] Samsung SmartSSD® Computational Storage Drive. https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html. Accessed: June 2024.

[13] vFPIO code. https://github.com/TUM-DSE/vFPIO.

[14] vitis-hls. https://github.com/Xilinx/HLS. Accessed: June 2024.

[15] Xilinx Merlin compiler. https://github.com/Xilinx/merlin-compiler. Accessed: June 2024.

[16] XRT and Vitis™Platform Overview. https://xilinx.github.io/XRT/master/html/platforms.html. Accessed: June 2024.

[17] Xup vitis network example (vnx). https://github.com/Xilinx/xup_vitis_network_example. Accessed: June 2024.

[18] 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.

[19] AMBA AXI-Stream Protocol Specification. https://developer.arm.com/documentation/ihi0051/latest/. Accessed: June 2024.

[20] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. Towards µs tail latency and terabit ethernet: Disaggregating the host network stack. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 767–779, New York, NY, USA, 2022. Association for Computing Machinery.

[21] 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.

[22] 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.

[23] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition.* O'Reilly Media, Inc., 2005.

[24] W Fu Fu and K Compton. An execution environment for reconfigurable computing(abstract only). In *International Symposium on Field Programmable Gate Arrays: Proceedings of the 2005 ACM/SIGDA 13 th international symposium on Field-programmable gate arrays*, volume 20, pages 267–267, 2005.

[25] Wenyin Fu and Katherine Compton. Scheduling Intervals for Reconfigurable Computing. In *2008 16th International Symposium on Field-Programmable Custom Computing Machines*, pages 87–96. IEEE, 2008.

[26] Marcel Gort and Jason Anderson. Design re-use for compile time reduction in FPGA high-level synthesis flows. In *2014 International Conference on Field-Programmable Technology (FPT)*, pages 4–11, 2014.

[27] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C Snoeren. SmartNIC performance isolation with FairNIC: Programmable networking for the cloud. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 681–693, 2020.

[28] 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.

[29] Zhenhao He, Dario Korolija, and Gustavo Alonso. Easynet: 100 gbps network for hls. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pages 197–203. IEEE, 2021.

[30] Joost Hoozemans, Johan Peltenburg, Fabian Nonnemacher, Akos Hadnagy, Zaid Al-Ars, and H. Peter Hofstee. FPGA Acceleration for Big Data Analytics: Challenges and Opportunities. *IEEE Circuits and Systems Magazine*, 21(2):30–47, Secondquarter 2021.

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

[32] 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.

[33] Robert Kirchgessner, Greg Stitt, Alan George, and Herman Lam. Virtualrc: a virtual FPGA platform for applications and tools portability. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 205–208, 2012.

[34] 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.

[35] Yi-Hsiang Lai, Ecenur Ustun, Shaojie Xiang, Zhenman Fang, Hongbo Rong, and Zhiru Zhang. Programming and synthesis for software-defined FPGA acceleration: status and future prospects. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 14(4):1–39, 2021.

[36] Joshua Landgraf, Matthew Giordano, Esther Yoon, and Christopher J. Rossbach. Reconfigurable Virtual Memory for FPGA-Driven I/O. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 556–571, New York, NY, USA, 2023. Association for Computing Machinery.

[37] Joshua Landgraf, Tiffany Yang, Will Lin, Christopher J Rossbach, and Eric Schkufza. Compiler-driven FPGA virtualization with SYNERGY. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 818–831, 2021.

[38] 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.

[39] 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.

[40] Mostafa W Numan, Braden J Phillips, Gavin S Puddy, and Katrina Falkner. Towards automatic high-level code deployment on reconfigurable platforms: A survey of high-level synthesis tools and toolchains. *IEEE Access*, 8:174692–174722, 2020.

[41] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, et al. {FlowBlaze}: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 531–548, 2019.

[42] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News*, 42(3):13–24, 2014.

[43] Siavash Rezaei, Eli Bozorgzadeh, and Kanghee Kim. UltraShare: FPGA-based Dynamic Accelerator Sharing and Allocation. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–5, 2019.

[44] Sloc Cloc and Code (scc). https://github.com/boyter/scc. Accessed: June 2024.

[45] Eric Schkufza, Michael Wei, and Christopher J Rossbach. Just-in-time compilation for Verilog: A new technique for improving the FPGA programming experience. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 271–286, 2019.

[46] Robert Schmid, Max Plauth, Lukas Wenzel, Felix Eberhardt, and Andreas Polze. Accessible near-storage computing with FPGAs. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–12, 2020.

[47] Robert Schmid, Max Plauth, Lukas Wenzel, Felix Eberhardt, and Andreas Polze. Accessible near-storage computing with fpgas. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[48] Jeffrey Shafer. I/O virtualization bottlenecks in cloud computing today. In *Proceedings of the 2nd conference on I/O virtualization*, pages 5–5. USENIX Association, 2010.

[49] David Sidler, Zsolt István, and Gustavo Alonso. Low-latency TCP/IP stack for data center applications. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2016.

[50] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. Strom: Smart remote memory. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[51] Gagandeep Singh, Mohammed Alser, Damla Senol Cali, Dionysios Diamantopoulos, Juan Gómez-Luna, Henk Corporaal, and Onur Mutlu. Fpga-based near-memory acceleration of modern data-intensive applications. *IEEE Micro*, 41(4):39–48, 2021.

[52] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 44–57, 2016.

[53] Hayden Kwok-Hay So and Robert Brodersen. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using borph. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(2):1–28, 2008.

[54] 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.

[55] Animesh Trivedi and Marco Spaziani Brunella. CPU-Free Computing: A Vision with a Blueprint. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, page 1–14, New York, NY, USA, 2023. Association for Computing Machinery.

[56] Harshavardhan Unnibhavi, David Cerdeira, Antonio Barbalace, Nuno Santos, and Pramod Bhatotia. Secure and Policy-Compliant Query Processing on Heterogeneous Computational Storage Architectures. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1462–1477, New York, NY, USA, 2022. Association for Computing Machinery.

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

[58] Mapping Kernel Ports to Memory: Vitis v2023.1 Unified Software Platform Documentation. https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Mapping-Kernel-Ports-to-Memory. Accessed: June 2024.

[59] 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.

[60] 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.

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

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

[63] Yue Zha and Jing Li. When application-specific ISA meets FPGAs: a multi-layer virtualization framework for heterogeneous cloud FPGAs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 123–134, 2021.

[64] Niansong Zhang, Xiang Chen, and Nachiket Kapre. Rapidlayout: Fast hard block placement of fpga-optimized systolic arrays using evolutionary algorithm. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 15(4):1–23, 2022.

[65] Jieru Zhao, Liang Feng, Sharad Sinha, Wei Zhang, Yun Liang, and Bingsheng He. Comba: A comprehensive model-based analysis framework for high level synthesis of real applications. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 430–437. IEEE, 2017.