Deploying Multi-tenant FPGAs within Linux-based Cloud Infrastructure

JOEL MANDEBI MBONGUE, DANIELLE TCHUINKOU KWADJO, ALEX SHUPING, and CHRISTOPHE BOBDA, University of Florida, USA

Cloud deployments now increasingly exploit Field-Programmable Gate Array (FPGA) accelerators as part of virtual instances. While cloud FPGAs are still essentially single-tenant, the growing demand for efficient hardware acceleration paves the way to FPGA multi-tenancy. It then becomes necessary to explore architectures, design flows, and resource management features that aim at exposing multi-tenant FPGAs to the cloud users. In this article, we discuss a hardware/software architecture that supports provisioning space-shared FPGAs in Kernel-based Virtual Machine (KVM) clouds. The proposed hardware/software architecture introduces an FPGA organization that improves hardware consolidation and support hardware elasticity with minimal data movement overhead. It also relies on VirtIO to decrease communication latency between hardware and software domains. Prototyping the proposed architecture with a Virtex UltraScale+ FPGA demonstrated near specification maximum frequency for on-chip data movement and high throughput in virtual instance access to hardware accelerators. We demonstrate similar performance compared to single-tenant deployment while increasing FPGA utilization, which is one of the goals of virtualization. Overall, our FPGA design achieved about 2× higher maximum frequency than the state of the art and a bandwidth reaching up to 28 Gbps on 32-bit data width.

CCS Concepts: • **Hardware → Reconfigurable logic and FPGAs**; *On-chip resource management*; Network on chip; • **Computer systems organization → Cloud computing;**

Additional Key Words and Phrases: Cloud, FPGA, multi-tenancy, network-on-chip, virtualization, KVM

ACM Reference format:
Joel Mandebi Mbongue, Danielle Tchuinkou Kwadjo, Alex Shuping, and Christophe Bobda. 2021. Deploying Multi-tenant FPGAs within Linux-based Cloud Infrastructure. *ACM Trans. Reconfigurable Technol. Syst.* 15, 2, Article 19 (November 2021), 31 pages.
https://doi.org/10.1145/3474058

1 INTRODUCTION

We have long passed the time when **Field-Programmable Gate Array (FPGA)** utilization was exclusively reserved to qualified and skillful hardware engineers. In fact, the continuous innovation in FPGA design flow and technology has opened the way to broader adoption of the reconfigurable paradigm. Software developers can now leverage **Computer-Aided Design (CAD)** tools

This work was partially funded by the National Science Foundation (NSF) under Grant CNS 2007320, and the Office of Naval Research (ONR) under the Grant CCN 0402-17643-21-0000.
Authors' address: J. M. Mbongue, D. T. Kwadjo, A. Shuping, and C. Bobda, University of Florida, 336A Larsen Hall Gainesville FL US 32611-6200; emails: {jmandebimbongue, dtchuinkoukwadjo, alexandershuping}@ufl.edu, cbobda@ece.ufl.edu.

1936-7406/2021/11-ART19 $15.00
https://doi.org/10.1145/3474058

such as Vivado **High-Level Synthesis (HLS)**, Vitis, and the Intel HLS Compiler to design high-performance hardware accelerators on FPGA for productivity improvement and with minimal quality of result loss compared to using hardware description languages. In addition, the increasing support for high-level programming languages such as Python and C/C++; and the emergence of tools such as RapidWright [39], that allow building crafted accelerators through Python and Java APIs, offer a wide range of possibilities for performance exploration in many domains such as artificial intelligence [37] and image processing [17].

While the industry previously focused on enabling developer-friendly CAD tools that can generate high-performance accelerators, using FPGAs within the stack of applications running in the cloud is emerging as the new rising trend. In fact, to cope with the versatile demand of applications in social media, content streaming, banking, shopping, and high-performance computing, data centers must accommodate heterogeneous architectures that can provide application-specific speedup. As a response, FPGA devices have been recently introduced in cloud infrastructure to provide acceleration for critical workloads in artificial intelligence, imaging, high-performance computing, and cryptography, with reduced power consumption. It is then no surprise to see major cloud providers such as Amazon, Baidu, Alibaba, and Microsoft now exploiting the low power consumption, high performance, and adaptability of FPGAs in their cloud portfolios [21, 24, 48, 54].

In this work, we propose a hardware/software architecture for transparent provisioning multi-tenant FPGAs within cloud infrastructure. Though FPGAs are now exposed to cloud users, current commercial FPGA-enabled cloud infrastructures have highlighted the lack of primitives and support allowing multiple workloads to space-share a single device. This could ultimately result in expensive cost of utilization considering the unit price of high-end accelerator cards (Xilinx Alveo U200 cards cost about $9000 [69]). As illustration, an **Amazon Web Service (AWS)** instance without FPGA can be about 8.5× cheaper than an equivalent with FPGA as of August 2020 [3, 4]. Another considerable issue is the waste of resources. In fact, FPGA devices most often gather more elements than what user workloads would typically need. As example, the Xilinx Virtex UltraScale+ FPGAs deployed within AWS F1 instances contain approximately 2.5 millions logic elements, 6,800 Digital Signal Processing (DSP) slices, and 75 Megabytes of block RAM (BRAM) [55]. After synthesizing an Advanced Encryption Standard (AES) core over 128 bits on that devices it uses about 0.39% of the chip area. This example illustrates that entirely assigning a high-end FPGA to a single user may outcome in significant hardware under-utilization. Considering that the capacity of integration in FPGA technology continuously increases as some devices now reach 9 millions of logic cells, it becomes a necessity to explore approaches to deploy multi-tenant FPGAs in the cloud.

The National Institute of Standards and Technology proposed several characteristics of cloud infrastructure.

Among these are (1) *on-demand self-service*: users can turn on/off virtual environments without intervention of the cloud service provider; (2) *broad network access*: the virtual environment can be accessed remotely over the network; (3) *resource pooling*: refers to the consolidation of hardware resources (storage, processing, memory, etc.) to serve users in a multi-tenant model with the goal of increasing hardware utilization; and (4) *rapid elasticity*: consists of allowing the provision and release of resources. It also encompasses scaling services with the demand [47]. Extending these concepts to cloud FPGAs could enable the simultaneous deployment of multiple accelerators on a single device and the scaling of FPGA resources with the user needs. This work primarily focuses on providing support for FPGA virtualization in Linux-based cloud infrastructure. Though the concepts developed in this work can apply to different types of **virtual machine monitors (VMM)**, we limit our study to **Kernel-based Virtual Machine (KVM)** and **Quick EMUlator (QEMU)** because of their increasing adoption in the cloud [9, 57]. We assume that the cloud characteristics

(1) and (2) above are inherently managed by the VMM and focus this research on *resource pooling* and *rapid elasticity*. Specifically, our contribution consists of the following:

(1) *Defining our FPGA Virtualization Concept:* We introduce and define elasticity and multi-tenancy with respect to the hardware/software virtualization architecture proposed in this work.
(2) *Extending a KVM infrastructure with FPGA management capabilities:* We propose a software architecture to support FPGA allocation, release, and access in KVM clouds.
(3) *Proposing an architecture that enables co-hosting workloads on a single device:* It explores the utilization of a **network-on-chip (NoC)** topology as support to FPGA multi-tenancy and hardware elasticity. It allows on-chip communication between workloads at about 1 GHz for data width between 64 and 256 bits. It also enables spatial and temporal sharing of FPGA logic between multiple cloud users with minimal performance degradation and resource overhead.

Addressing the security challenges that may arise from FPGA multi-tenancy in the cloud is out of the scope of this work.

The rest of the article is structured as follows: Section 2 reviews recent utilization cases of FPGAs in the cloud, both from industry and academia. Next, Section 3 discusses background concepts that are necessary to understanding the proposed FPGA virtualization architecture. Section 4 elaborates on the different components that enable exposing multi-tenant FPGAs with the virtualization stack of linux-based cloud infrastructures. Afterwards, experimental results are presented in Section 5 and Section 6 concludes the article.

2 FPGAS IN THE CLOUD

2.1 FPGA Utilization in the Cloud

The utilization of FPGAs in the cloud is a rising trend. In recent years, an increasing number of **cloud service providers (CSP)** have started integrating FPGAs in their investment portfolio. The FPGA utilization model in the cloud varies depending on the CSP objectives as shown in Table 1. FPGAs can be used as hardware accelerators for cloud software and services (Software as a service and Platform as a service models). For instance, Microsoft started speeding up the Bing search engine's ranking algorithm using Intel Stratix V FPGAs in 2016. FPGAs are also delivered as part of virtual instances within cloud infrastructures (Infrastructure as a service or IaaS model). AWS provides an example of such utilization of FPGAs. AWS provisions cloud virtual instances enabled with custom FPGA tool chain and library of IPs, allowing developers to design, compile and deploy crafted accelerators on FPGAs in the cloud. Some CSPs such as PLUNIFY do not provision FPGA resources, but provide services to decrease compilation times. Our work addresses the virtualization of FPGA resources in IaaS.

2.2 Literature Review

The virtualization of FPGA resources in the cloud is an active field of research that has seen multiple contributions in recent years. Some works present solutions to the temporal allocation of FPGA kernels [1, 19, 66, 78]. They explore techniques to successively allocate full or partial FPGAs to cloud tenants over time. Other contributions describe architectures that expose FPGA regions to cloud tenants through the IaaS virtualization stack. Table 2 presents a list of such published research.

It mainly classifies the architectures based on the shell area overhead, the number of virtual regions per FPGA, whether multi-tenancy is enabled or not, the presence or absence of on-chip

Table 1. List of Major FPGA Cloud Providers

Providers	Country	FPGA Utilization Model	VMM	Maker	FPGA	Year
Microsoft [48]	USA	FPGAs accelerate background tasks in the networking, web search ranking on Bing, and AI inference	Hyper-V	Intel	Stratix V, Stratix 10, Arria 10	2014
Nimbix [50, 51]	USA	VMs are provisioned with FPGA development tools, and can program and communicate with FPGA devices	—	Xilinx, Intel	Alveo U50, U200, U250, U280	2014
IBM Cloud [32]	USA		—	Xilinx	7-series, Kintex UltraScale+	2015
Amazon [5]	USA		Xen, KVM	Xilinx	Virtex UltraScale +	2016
Baidu [8, 61]	China		KVM	Xilinx	Kintex UltraScale	2016
Alibaba Cloud [2, 13]	China		Xen, KVM	Xilinx, Intel	Virtex UltraScale+, Arria 10	2017
Huawei [30, 31]	China		Xen, KVM, VMware ESXi	Xilinx	Virtex UltraScale+	2017
OVHcloud [11, 26]	France	Use FPGAs in the network stack to mitigate denial-of-service attacks on the cloud infrastructure. VMs running on OpenStack are also provisioned with FPGA developments tools and access to FPGA devices	VMware	Xilinx, Intel	UltraScale+, Stratix V	2017
PLUNIFY [71]	Singapore	Runs design-space exploration with machine learning in the cloud for timing closure and reduced design iteration.	—	Xilinx, Intel	Cyclone series, Arria II, Stratix IV, Arria V, Stratix V, Spartan 6, 7, Artix7, Zynq, Virtex 7, Kintex 7, UltraScale, UltraScale+	2018
Texas Advanced Computing Center [63]	USA	Investigate use cases for FPGAs in data centers in partnership with Microsoft	—	Xilinx, Intel	Stratix V, Virtex 7	2018
VMWare [18]	USA	Collaborate with partners to enable FPGA management capabilities on vSphere	VMware ESXi	Xilinx	Alveo U250	2020

Table 2. Study of Recent Research in FPGA Virtualization Architecture for Cloud Infrastructure

Published research	Shell area overhead	# Virtual regions/FPGA	Spatial Sharing	On-chip comm. support	Data Width (bits)	Fmax (MHz)	Access Method	PCIe version	Network specification
Fahmy et al. [22]	7%	4	Yes	No	256	250	PCIe	Gen 3 × 8	—
Weerasinghe et al. [68]	21.7%	1	No	No	64	156.25	Network	—	10GbE
Tarafdar et al. [65]	27%	1	No	No	43	125	Network	—	10GbE
Mbongue et al. [43]	1%	4	Yes	Yes	—	227	PCIe	Gen3 × 16	—
Catapult [56]	23%	1	No	No	16-48	200	PCIe & Network	—	10Gb SAS
Byma et al. [12]	19%	—	Yes	No	256	160	Network	—	10GbE
Feniks [77]	13%	—	Yes	No	—	—	PCIe	Gen3 × 8	—
Mandebi et al. [46]	0.1%	6	Yes	Yes	32–256	1500	PCIe & Network	—	Fast Ethernet
Chen et al. [14]	6.46%	4	Yes	No	—	100	PCIe	Gen2 × 8	—
Asiatici et al. [6]	—	3	Yes	No	—	—	PCIe	—	—

communication support, the data width, the F_{max}, and the access method to the virtual resources on the FPGA. The number of virtual regions per FPGA does not represent the maximum number of accelerators that can be hosted in a single device but reflects the results reported in the experimental evaluation. For instance, Chen et al. [14] divide each FPGA into four locations or "virtual FPGA." The architecture enables multi-tenancy and provisions hardware resources over PCIe. However, this architecture is limited by two main factors: It only allows the use of pre-built hardware functions, and it does not support direct on-chip communication. Not supporting on-chip communication imposes middleware copy for data movement between the accelerators of a user. To minimize the data movement overhead, an on-chip interconnect can be used between virtualized hardware regions [43, 46]. Weerasinghe et al. [68] observe that network-attached FPGAs may offer lower latency and higher throughput compared to accessing accelerators over PCIe. However, they did not provide details on their PCIe interface, which prevents assessing the baseline of the

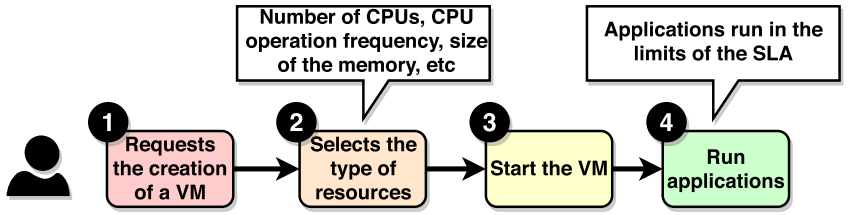

Fig. 1. Illustration of the general flow for creating virtual machines in cloud infrastructures.

comparative study. The works from Weerasinghe, Tarafdar, and Catapult [56, 65, 68] do not support spatial sharing of FPGA components between the workloads of multiple cloud users—they focus on FPGA time sharing. Moreover, there have been research that explore FPGA multi-tenancy considering memory and network virtualization. Rozhko proposed an FPGA virtualization architecture that provides co-hosted hardware accelerators with AXI ports to access shared network interfaces and dedicated memory space [58]. Chiotakis et al. detail a framework that optimizes the utilization of FPGA resource for **network function virtualization (NFV)** [15]. The framework mitigates the lack of flexiblity of single root input/output (IO) virtualization that does not enable runtime re-allocation of virtual functions to tenants. Therefore, vFPGAmanager is designed to support virtual function time-sharing through runtime reconfiguration of the switch interconnect interfacing hardware accelerators with the memory space of the PCIe-attached FPGA device. Nobach et al. demonstrate that the utilization of FPGAs to implement NFV results in significant performance improvement. In addition, about 39% cost reduction is observed due to the runtime reprogrammability of FPGAs [52]. Mbongue et al. study the performance that can be achieved in multi-tenant cloud FPGAs [45].

3 BACKGROUND

In this section, we discuss key concepts that we believe are necessary to the full understanding of the proposed solution.

3.1 FPGA Multi-Tenancy

Given that elasticity requires that all acquired resources are ultimately released at some point in time, we focus our multi-tenancy study on FPGA sharing in the space domain. Specifically, this work seeks to enable spatial sharing of a single FPGA device between multiple cloud tenants. We consider the resource access model of traditional cloud infrastructures without FPGA support presented in Figure 1. It typically starts with a user request to a CSP to set up a VM. The user first selects the type and amount of resources to attach to the VM such as Central Processing Unit (CPU), Random Access Memory(RAM), storage, and boots the virtual environment and then starts running applications. Tasks can run as long as they do not violate the **Service-Level Agreement (SLA)**. For instance, if a VM is set up with a disk of 1 TB, then it will not be possible to store more data without requesting additional storage. This flow is generally adopted in cloud infrastructures delivering VMs. We utilize a similar flow to expose FPGA components to the cloud users. The approach that we propose enables selecting "*FPGA unit of virtualization*" as part of VMs. Given the physical layout of FPGA chips (array of logic components and interconnect), each "*FPGA unit of virtualization*" will represent a designated area on the device that we call "*virtual region*" or VR. We therefore propose an approach in which VRs are advertised in the cloud as opposed to entire FPGAs (see Figure 2). The amount of resources available within each VR is defined by the CSP as it is the case with the unit of memory, storage, and processing offered in VM flavors. This

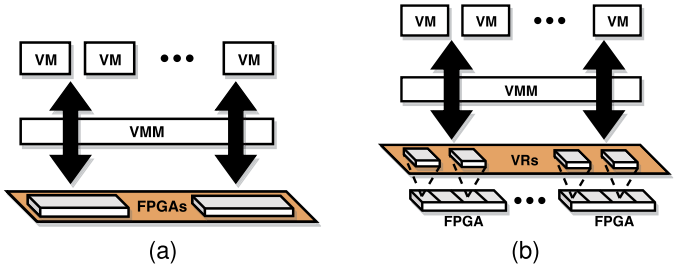

Fig. 2. Illustration of the provisioning models. (a) Model in which entire FPGAs are advertised in the cloud. (b) Proposed model: It provisions FPGA regions in the cloud.

work does not aim to define how to divide a device into VRs, but proposes a flexible architecture that allows CSPs to configure FPGA sharing to improve hardware consolidation. Since the VRs in a VM domain may exchange data to pipeline the execution of workloads, we implement a soft-NoC interconnect in the Shell as a support to hardware elasticity. Our concept of elasticity differs from that of Vaishnav et al. [67], as we consider a model in which users fully control (runtime programming through partial reconfiguration) units of FPGA assigned to their domains by the cloud infrastructure.

3.2 Common Hardware Virtualization

3.2.1 CPU Virtualization. CPU virtualization is well investigated and currently relies either on instruction set architecture translation or hardware support for efficient execution of problematic instructions (instructions that fail without faulting in user mode) such as PUSH in ×86 architectures [49]. However, FPGAs do not follow the Von Neumann architecture. Rather than executing a sequence of well-defined instructions, FPGAs allow the implementation of arbitrary RTL logic—most CPU virtualization techniques are not applicable to FPGAs.

3.2.2 Memory and Disk Virtualization. Memory and disk virtualization is fundamentally based on reserving specific memory and disk locations for VM data [62]. Though FPGAs are intrinsically a set of blocks like memory and disk units, the FPGA blocks are of different types such as Look Up Tables (LUT), Flip Flops (FF), etc., and connect into custom circuits, requiring a different type of virtualization than that of the memory hierarchy. Finally, FPGAs could fall into the category of IO devices, since they are most often added as co-processors into systems. Yet, their reconfigurability makes their functionality unpredictable as it can be changed at runtime, therefore requiring middleware management beyond IO access (runtime reconfiguration, off-chip memory access, etc.). Because we seek efficient FPGA multi-tenancy in the cloud, we leverage the state-of-the-art research in memory and IO virtualization. We start by dividing the FPGA fabric into disjoint VRs purposed to host VM workloads, and propose an architecture enabling fast IO access to VR registers.

3.2.3 IO Virtualization. IO virtualization can be achieved either by software (emulation and paravirtualization) or hardware (directIO, single/multiple root IO virtualization) means as illustrated in Figure 3. The first software solution is called "emulation" (see Figure 3(a)). In this approach, each attempt to execute an IO instruction raises a system call that is trapped and executed by the VMM in privileged mode. While this approach has the benefit of not requiring **guest operating system (GOS)** modification, it incurs high overhead because of the recurrent context switches between privileged and non-privileged modes. "Paravirtualization" prevents context switches by implementing communication between "*frontend drivers*" in the VM and "*backend drivers*" in the VMM (see Figure 3(b)) [7]. Though it modifies the GOS, it improves security and platform stability as the hardware is accessed through an unified driver in the VMM [23]. Despite optimizations,

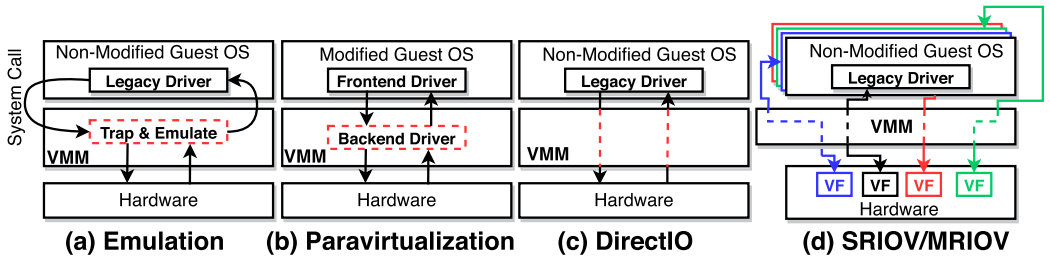

Fig. 3. Summary of the state-of-the-art IO virtualization approaches.

software-based approaches incur performance penalties from the VMM interference. As a solution, "directIO" removes VMM overhead by directly assigning the IO device to the VM (see Figure 3(c)). Though it allows device IO speed, it does not enable multi-tenancy, which is one of the core reasons for virtualizing hardware. "Single/Multiple root IO virtualization (SRIOV/MRIOV)" extend the directIO concept with device sharing capabilities. They provision a set of **"virtual functions" (VF)** to VMs that can all access the same device through the PCIe interface (see Figure 3(d)). Though SRIOV/MRIOV achieve near directIO performance [20, 75], they do not authorize runtime remapping as the allocation of VFs is persistent for the entire lifecycle of VMs. This is not ideal in an environment that aims to implement elasticity: it should be possible to allocate, de-allocate, and re-assign FPGA regions to cloud tenants at runtime.

In this work, we consider FPGAs as IO devices and propose a hardware/software architecture that extends the state-of-the-art IO virtualization approaches. We implement an approach that, just as in SRIOV and MRIOV, hosts several VFs (called virtual regions or VRs in the context of our work) on a single FPGA device. As opposed to SRIOV and MRIOV, we implement a paravirtualized architecture in the KVM hypervisor that enables runtime remapping of FPGA resources and rapid elasticity. The following sections will detail our proposed FPGA virtualization architecture.

3.3 Hardware Elasticity on FPGA

Service elasticity generally consists in allowing the provision and release of resources, as well as scaling the capacity with the needs [47]. Provisioning elastic cloud FPGAs entails allowing developers to program the FPGAs in their domain with designs of different hardware footprints. It is already possible in cloud infrastructures that expose single-tenant FPGAs to the users such as AWS F1. However, this is not entirely applicable to space-shared FPGAs. The reason is, a cloud user cannot dispose of the entire fabric as in the case of single-tenant FPGAs. To space-share a device, the fabric is divided into logically isolated regions and partial reconfiguration is used to reprogram specific FPGA regions at runtime. Therefore, the CSP pre-defines flooplanning constraints before provisioning FPGAs in the cloud. However, such floorplanning cannot be modified at runtime. As a result, if a user design cannot fit into the FPGA regions that are available on a cloud FPGA, then several solutions could be adopted among which: (1) look up on other FPGAs, (2) migrate running workloads to other FPGAs or FPGA regions to free space, and (3) divide the design into smaller modules. Implementing solution (1) may not be sufficient as it could result in the same issue. The application of solution (2) in the context of FPGAs faces some difficulties. Migrating workloads in cloud and virtualization systems transfers the execution environment of an application to a new host while attempting to minimize the down time [76]. However, FPGA accelerators are compiled to generate circuits that generally depend on timing, floorplanning, and power constraints. As a result, a hardware design may not be portable to other FPGA regions or other FPGAs. Furthermore, CPUs operate in a well-known sequence summarized in fetching and decoding instructions, loading operands, executing, and performing memory accesses [29].

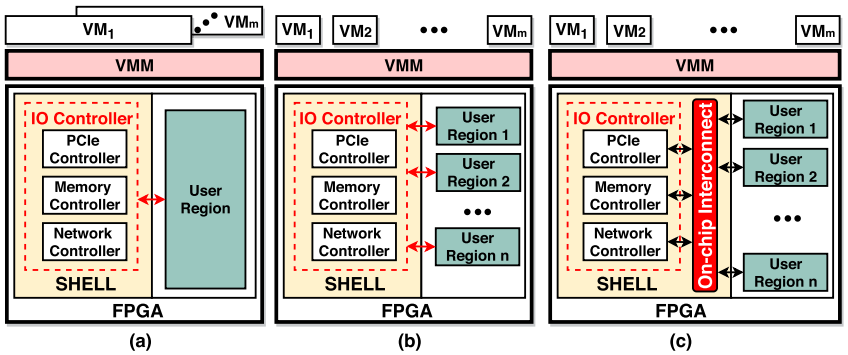

Fig. 4. Overview of typical shell architectures within clouds provisioning FPGA resources. (a) The FPGA is single-tenant and is successively allocated to workloads from different tenants [56, 65, 68]. (b) The FPGA is multi-tenant and is space-shared between multiple tenants [6, 12, 14, 22, 77]. (c) Multi-tenant FPGA model that extends the models presented in (b) with an on-chip communication support between the virtual regions [43, 46].

However, FPGA designs can implement any custom function, making it difficult to know when to transfer the hardware execution and virtually impossible to see the content of internal registers. Therefore, properly performing the migration without compromising the design functionality remains challenging. In this work, we combine solutions (1) and (3). In our hardware elasticity model, the FPGA is divided into multiple regions. The VMM can then manage pools of VRs across the FPGAs as illustrated in Figure 2(b). Given that a hardware design can be distributed across several VRs, communication channels should enable data movement at the hardware level. Therefore, our virtualization architecture comprises an on-chip interconnect that provides low latency communication lines between the VRs hosted on an FPGA to support hardware elasticity.

4 PROPOSED FPGA VIRTUALIZATION IN KVM

4.1 Architecture Overview

Our proposed virtualization scheme is a hardware/software architecture. It spans from the software layer in the GOS down to the FPGA devices attached to the hosts. It provides mechanisms to capture hardware accesses in the guest and transfer the requests to the actual hardware. It also augments KVM hosts with FPGA management utilities and provides hardware-level support for FPGA multi-tenancy. The proposed virtualization architecture comprises three major layers:

- A *shell layer* that implements static components on FPGA to support hardware multi-tenancy and elasticity.
- An *FPGA management layer* that extends KVM hosts with FPGA management capabilities.
- A *communication layer* that defines the communication between guest applications and hardware accelerators on FPGAs.

The following sections will successively describe and elaborate on each of the layers.

4.2 Shell Layer

The shell layer is a set of static components on the FPGA that cannot be modified by cloud users. These components form the communication infrastructure that enables on- and off-chip connectivity; they also assist in VR provisioning. Figure 4 provides an overview of the typical cloud shells. It also highlights the portion of the design that goes into the shell. We implement

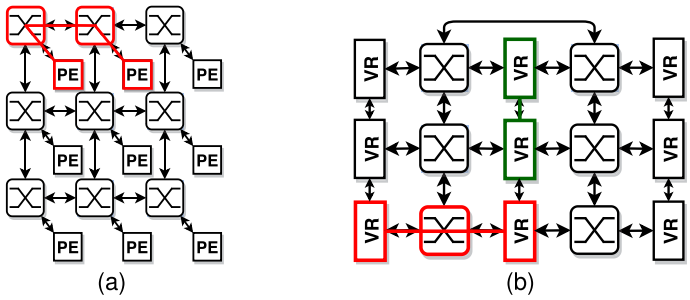

Fig. 5. (a) Traditional 2D bidirectional mesh architecture. (b) Our proposed NoC topology. It reduces the radix of routers and enables direct inter-VR communication.

the model illustrated in Figure 4(c). The shell is made of two major components: *(1) IO Controllers,* to manage the communication with off-chip resources such as memory, CPU, and so on. In this work, we do not elaborate on the interfacing logic of the shell as we rely on vendor IPs to design high-performance IO controllers. *(2) On-chip Interconnect,* which implements a soft-NoC topology that enables efficient on-chip communication between VRs to support hardware elasticity.

As an observation, large VRs may lead to resource under-utilization if allocated to smaller user designs. However, smaller VRs could result in user designs broken into multiple blocks, potentially increasing data movement overhead. It this therefore necessary to combine the FPGA division into VRs with a placement algorithm that will minimize the resource waste and decrease on-chip communication overhead. As mentioned earlier, this work aims not to define an optimal division of the FPGA into VRs. We focus on proposing an architecture that provides CSPs with flexible provisioning options.

4.2.1 Proposed Soft-NoC Topology. To efficiently implement *resource pooling*, we seek to maximize the number of concurrent workloads that a single device can host. In other words, we aim to design a shell that minimizes the resource overhead on the FPGA. We focus our architecture optimization on the NoC, as efficient IO controllers such as PCIe blocks are already well supported by vendor IPs. Further, for fast data movement between VRs, the NoC should achieve near device specification F_{max}—this means that the number of sequentially connected LUTs within the architecture should be minimized.

There are several NoC topologies, such as ring, star, hypercube, and so on. We consider a two-dimensional (2D) Mesh style for our soft-NoC. Mesh topologies usually feature **processing elements (PE)** with a network interface attached to a router. Architectures implementing a 2D mesh typically have routers with 5 interfaces (4 interfaces to communicate with adjacent routers and 1 interface connected to a PE). Figure 5(a) illustrates a general view of a 3×3 2D mesh. Mesh topologies have two defects in terms of the FPGA logic needed for each router and the overall communication latency. (1) A smaller network diameter is tightly coupled to a larger router radix (number of IO ports of the router). This allows reaching destinations in a few hops from any source and can possibly reduce communication overhead. However, crossbars and allocators are well known to grow quadratically in logic with the radix of the routers, resulting in substantial routing delays, lower operating frequency, and higher area and power consumption. (2) In a mesh, each router serves a single PE. This means that any communication between PEs requires a minimum of two hops, each router introducing potential delays depending on the traffic. Because we target lower resource utilization, high frequency of operation, and low communication latency, we propose the topology illustrated in Figure 5(b). It implements a mesh topology in which routers have at most 4

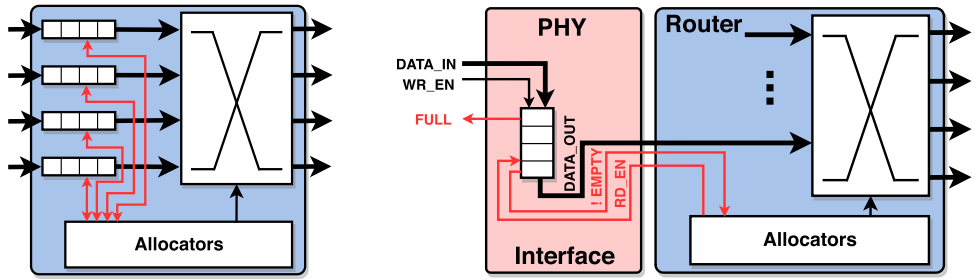

Fig. 6. Router optimization for F_{max} improvement and area reduction.

ports. As opposed to a regular mesh, each router is connected to 2 VRs, which decreases the number of hops. To keep the radix of the routers to 4 with 2 VRs connected, we reduce the dimension of the routing. Packets are either pushed up/down or injected into the VRs. We also enable direct communication links between adjacent VRs, allowing us to offload the routers and stream data every clock cycle between workloads. The proposed NoC architecture uses 1.5× less routers than a traditional mesh, reducing the area and power overhead.

We leverage architecture optimization in high-end FPGAs to maximize the NoC operating frequency while reducing the area and power consumption. For instance, UltraScale devices are arranged in a column-and-grid layout of clock regions that are 60 CLBs in height. A CLB contains eight 6-LUTs and 16 flip-flops. This high capacity of integration allows packing the NoC routers over a few CLBs (<%1 of the chip). In addition, rapid signal transmission is made possible by the abundance of switches and long wires spanning 16 CLBs [70]. With fabric switches connecting large datapaths, the NoC can implement high frequency wide buses. We use placement constraints to force NoC into specific areas of the chip and prevent CAD tools from using more CLBs than necessary. Next, we constrain routing within the boundaries of the NoC allocated areas, freeing up more resources for user designs. Our NoC implementation uses the AXI4 interfaces for standardization. Though our topology may lead to higher hops compared to a traditional mesh in some cases, its higher connectivity between VRs offers more flexible placement options.

4.2.2 *Router Component.*

Architecture: In this section, we discuss design choices and optimization on the router's internal architecture.

We start with the typical bidirectional router architecture that is presented in Figure 6(a). The *Input Buffers* serve two purposes: (1) minimizing the occurrence of metastability between VR and router clock domains and (2) temporary data storage when the destination is not ready. To route traffic to the right destination, each router implements a *Crossbar Matrix* that connects input and output channels and allows parallel data streaming. We optimize the size of the crossbar by removing unnecessary muxes. In fact, if we consider that we have n inputs and m outputs, then each of the output lines only needs $n - 1$ switches, since a VR will not send data to itself. Each router therefore has $(n - 1) \times m$ switches in the crossbar instead of $n \times m$. With four-port routers, each line in the crossbar thus multiplexes three entries. In our topology, the first and last routers only need three interfaces (see Figure 5(b)). This is simply a consequence of the absence of a fourth component to attach. Since one of the goals is to keep a low hardware footprint, we implement a three-port version of the router. This reduces the number of switches to 2 on each line of the crossbar. It also gives cloud providers the flexibility to assemble the topology that meets their needs by combining routers with three and four interfaces.

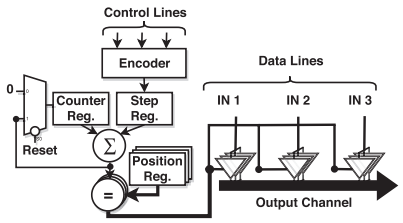

Fig. 7. Mutual Exclusion Logic.

INPUT 0	INPUT 1	STEP	SELECT
0	0	Z	Z
0	1	0	1
1	0	0	0
1	1	1	0 + step

Fig. 8. 2-Input Encoder.

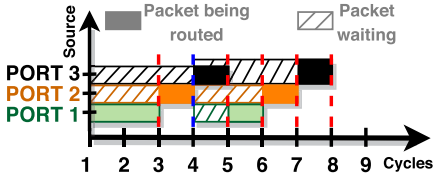

Fig. 9. Illustration of the mutual exclusion when packets at destination of Port 4 of arrive simultaneously from Port 1, Port 2, and Port 3 in a four-port router.

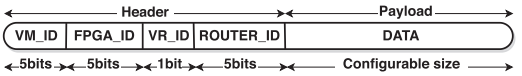

Fig. 10. Packet structure.

Kapre et al. [34] observed that buffers can increase router resources by 20–40%, which comes at the cost of area, delay, and power. As in Hoplite [34], we therefore implement bufferless routers as illustrated in Figure 6(b). We remove the buffers from the routers and keep data at the interface until the router is ready to process the packets. The *Allocators* are responsible for loading the data into the crossbar. Each allocator monitors a specific channel of the crossbar and implements a 3-way handshake protocol that works as follows: (1) The VR lets the allocator know that data are available through the buffer "EMPTY" signal. (2) When the crossbar is ready, the allocator pulls the data by asserting the "RD_EN" signal. (3) The data are loaded in the crossbar. Each allocator is also responsible for mutual exclusion between packets that pass through the same crossbar output channels. The purpose is to make sure that only one packet crosses an output channel at a time. Figure 7 summarizes the mutual exclusion logic. Based on the control lines asserted that signals the presence of incoming packets, an encoder determines the packet that is read in. If there are multiple packets from different sources, then only one packet is pulled from an input interface at a time to establish fairness. Figure 8 shows the logic of a two-input encoder. To illustrate the management of mutual exclusion, consider a four-port router with traffic coming from ports 1, 2, and 3 to port 4. Figure 9 summarizes how the *allocator* loads the packets. In cycle 1, there is incoming traffic from the three ports. The packets are routed one at a time. In cycle 4, when new data arrive at the 3 input ports, the data are loaded in the same way. From the third cycle, data will simply keep flowing out of the router, because the inputs are pipelined.

Routing Algorithm: Each router parses the header of an incoming packet to decide the next hop. The packet structure is presented in Figure 10. The header has a fixed width of 16 bits, and the payload has a configurable size. The destination of a packet is defined by the combination of the VR_ID (VR identifier) and ROUTER_ID (router identifier). Details on the VM_ID (VM identifier) and FPGA_ID (FPGA identifier) will be provided in Section 4.3. We limit the width of the header to 2 bytes as it may not be the case that a single device will be divided into a high number of VRs. For instance, the ROUTER_ID is represented over 5 bits. In other words, there can be a maximum

ALGORITHM 1: Packet Routing

```
1:  Input: incomingPacket, routerId
2:
3:  for each incomingPacket do
4:      if (getRouterID(incomingPacket) > routerId) then
5:          forwardToNorth(incomingPacket);
6:          goto Next;
7:      end if
8:      if (getRouterID(incomingPacket) < routerId) then
9:          forwardToSouth(incomingPacket);
10:         goto Next;
11:     end if
12:     if (getVRID(incomingPacket) == 0) then
13:         forwardToWest(incomingPacket);
14:     else
15:         forwardToEast(incomingPacket);
16:     end if
17:     Next:
18: end for
```

of 32 routers on a single device, which corresponds to at most 64 VRs. However, these numbers are parameters that can be modified by the CSP.

Although we opted for bufferless routers like Hoplite does, we do not implement deflection for two reasons. First, it may lead to an unpredictable number of hops. Second, the routers of our topology only route in one dimension. As a result, packets are either injected into one of the VRs connected to the router, or pushed up or down to the next router depending on the destination address. The VR_ID is represented on 1 bit. It identifies the VR that is the destination of the packet. Since each router is connected to at most 2 VRs (west and east sides), a VR_ID that is equal to 0 corresponds to the west VR, and a VR_ID that is 1 refers to the east VR. The ROUTER_ID labels the router to which the destination VR is connected. Algorithm 1 summarizes the routing procedure. It first checks the ROUTER_ID. If the current ROUTER_ID is greater (respectively, smaller) than that of the packet being transmitted, then the packet is pushed up (respectively, pushed down). If the packet has reached the destination router, then the VR_ID field is checked to determine the VR into which the packet will be injected.

4.2.3 Virtual Region Architecture. The architecture of FPGA provisioned regions is illustrated in Figure 11. The largest component of the VRs is the *USER REGION*. It hosts the cloud user's custom designs and implements the partial reconfiguration paradigm. The *USER REGION* is the only component of the VR that the cloud user edits; the VMM configures the other ones. The VRs also feature an *Access Monitor*, which only accepts packets from authorized VMs. It comprises a *Validity Checker* that receives the incoming packets from the input queue. It checks if the VM_ID contained in the packet header matches with an entry in the **Access Vector Cache (AVC)**. The list of authorized VMs is loaded in the AVC by the VMM at configuration time. In general, the AVC will contain a single entry, but we allow multiple VMs to share an accelerator. This feature could support designing services that span beyond the domain of a single VM. The *Access Monitor* removes the packet header and only forwards the payload to the *USER REGION*. The user designs only receive the payloads to prevent malicious applications from trying to access resources out of their domain. Developers are simply provided with an AXI-stream interface to implement accelerators. In addition to allowing users the program the FPGA at runtime, the USER REGION

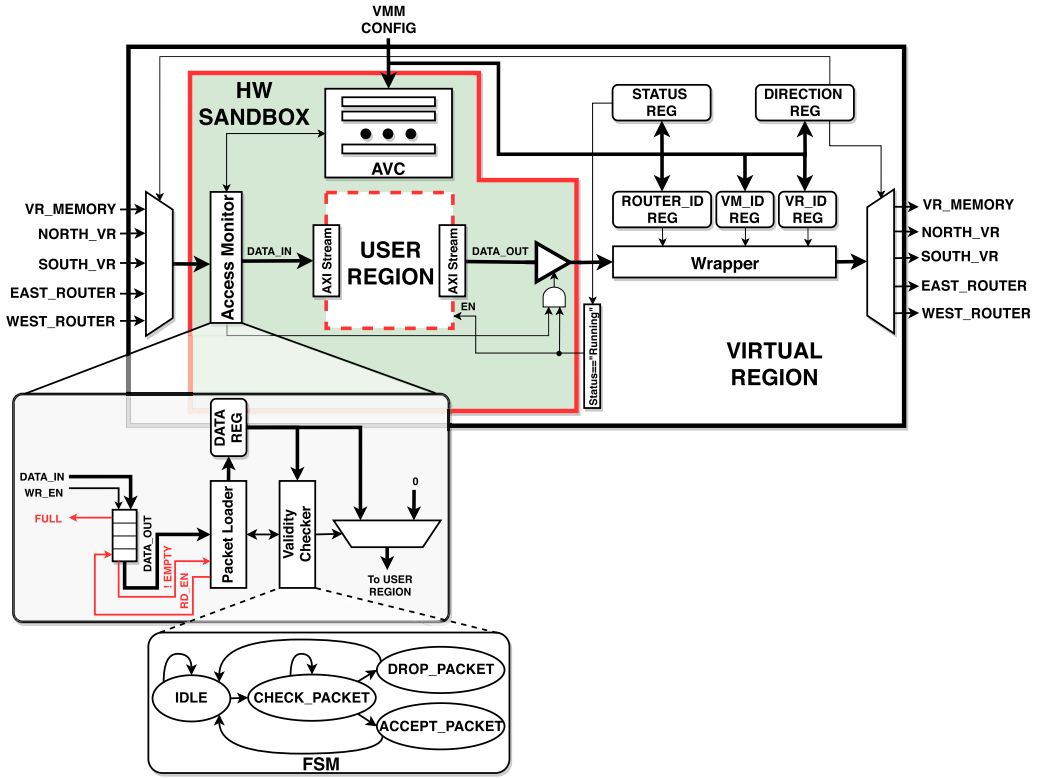

Fig. 11. Architecture of a Virtual Region.

abstracts away hardware detail related to the cloud infrastructure. Therefore, user designs only implements AXI-Stream interfaces that is directly connect to another VR or memory unit. At configuration time, the VMM edits the content of the VR registers. If the VR communicates with other FPGA regions, then the router and VR identifiers of the destination are stored in the *ROUTER_ID* and *VR_ID* registers. The VM identifier is also written into the *VM_ID* register. Whenever a VR is sending a packet out, the *USER REGION* produces the payload that is appended to the header generated in the *Wrapper* module to form a valid packet. By pre-configuring VR registers before running hardware accelerators, we prevent cloud users from send packet to an unauthorized destination, which could disrupt the normal execution of other co-hosted designs. The status register *STATUS REG* defines the current state of a VR. Table 3 summarizes the possible states of a VR.

We consider eight different states that are encoded on 3 bits. A VR is marked "Available" when it is not assigned or considered to be assigned to a VM. The "Reserved" state identifies a VR that is considered as potential host to a VM workload. Therefore, the VR cannot be considered in the resource allocation of any other VM to prevent a situation in which a VR will be allocated to multiple VMs. When a VR is allocated to a VM, its status is set to "Allocated." After the user design is programmed into the *USER REGION* and the VR registers are edited, the status of the VR is changed to "Programmed." The "Running" state is set when the VR starts running user workloads. The "Suspended" state provides a control to the VMM to prevent a VR from receiving and emitting traffic. It could be useful if any issue such as the violation of an SLA requires temporarily suspending a VR. If additional IO operations, such as populating memory, are needed before a running hardware task, then the VR is set in the "Waiting" state. Finally, the "Deallocated" state is a transition before

Table 3. Encoding of the States of a VR

3-bit Code	State	Description
000	**Available**	The VR is in the pool of available resources
001	**Reserved**	The VR is being considered in an allocation phase. It cannot therefore be seen as available
010	**Allocated**	The VR has been assigned to a specific VM domain
011	**Programmed**	The logic implemented in the VR has been programmed through partial reconfiguration
100	**Running**	The VR is running the hardware workload from a VM
101	**Suspended**	The operation of a VR is frozen
110	**Waiting**	The VR is waiting for specific external operation before proceeding with running hardware workloads
111	**Deallocated**	The VR is being prepared to be added into the pool of resources that are available

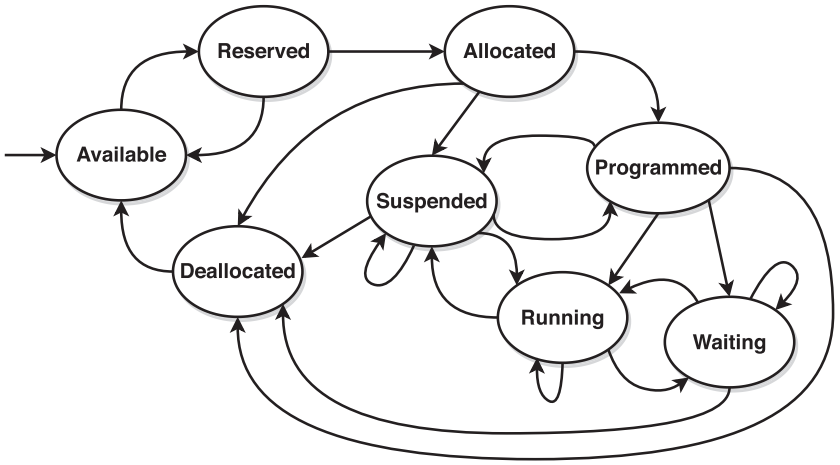

Fig. 12. Transitions between the different states of a VR.

making the VR available to other VM. This transition is used to clear the internal registers and memory of a VR. The transitions between the different states of a VR is illustrated in Figure 12. Starting with the "Available" state, a VR can only become "Reserved." Next, it can transition to the "Allocated" state or return in the pool of available resources. From the "Allocated" state, a VR can become "Programmed," "Suspended," or "Deallocated." After being "Programmed," a VR can be "Suspended," "Running," or "Waiting." After executing a hardware task, a VR is first "Suspended" before being "Deallocated" and "Available." It prevents the emission of any other traffic after the completion of a task.

After the discussion on the hardware components that are embedded in the shell, the following section addresses the management of FPGA resources at the level of the VMM.

4.3 FPGA Management Layer

Figure 13 illustrates the overall hardware/software virtualization architecture. It shows an overview of the components on FPGA, as well as the software virtualization stack. The FPGA is divided into VRs that are accessed through a hardware sandbox to perform access control. Figure 13 also presents a high-level view on FPGA management features added to KVM hosts: a resource allocation module and communication components (communication components are presented in

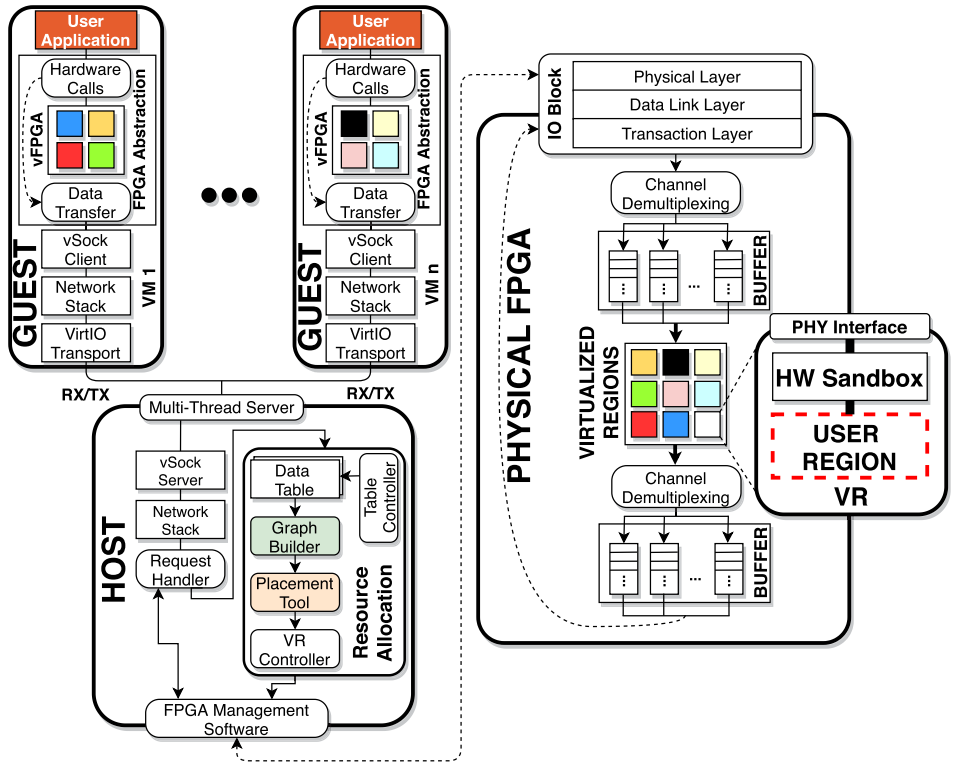

Fig. 13. Hardware and software components in the proposed FPGA virtualization stack.

Section 4.4). Each VM is presented with an abstraction of FPGAs in which each "*hardware call*" to an accelerator is replaced by a "*data transfer*" to the actual hardware. In the host, the *Request Handler* is responsible from checking the type of the incoming requests, which could be (i) a read operation, (ii) a write operation, (iii) a request for VR allocation, and (iv) a request for releasing a VR. In the event of IO requests to FPGAs (types (i) and (ii)), the *Request Handler* simply forwards the traffic to the *FPGA Management Software* that interfaces with the hardware. The functionality of the *FPGA Management Software* is described in Section 4.4.2. The *Resource Allocation* module manages the lifecyle of VRs in VM domains. It comprises two data tables that are the *VR/FPGA Table* and the *CID/VR Table*. The *VR/FPGA Table* establishes the correspondence between the VRs and the FPGA from which they are provisioned. The *CID/VR Table* keeps the correspondence between each VM (identified by a 32-bit *context identifier* or CID) and their active VRs using the CID. The *Table Controller* allows updating the status (available, reserved, etc.) of the VRs in the data tables. The *Graph Builder* updates the data structure that keeps the list of available VRs in memory. The *Placement Tool* is the component in charge of allocating VRs to VMs in a way that minimizes the data movement overhead between communicating hardware accelerators. The *VR Controller* is in charge of programming FPGA devices using partial reconfiguration and initializing VR internal registers at configuration time.

Data Structure Definition: When the VMM is solicited to acquire access to FPGA resources (type (iii) request), the *Request Handler* invokes the *Resource Allocation* module that runs the admission control strategy. It essentially performs three main functions:

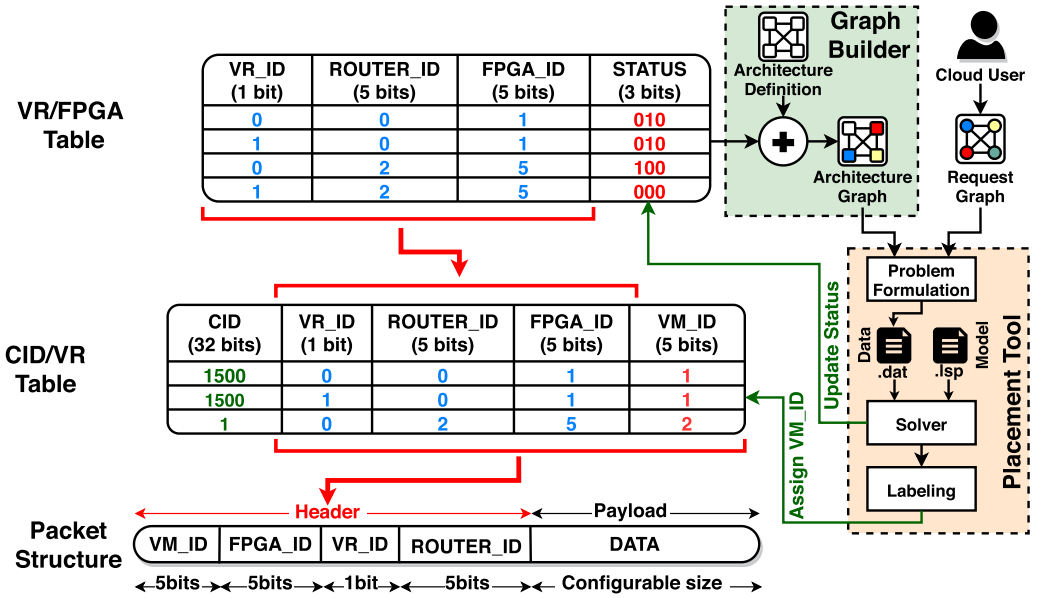

Fig. 14. Data Tables and Resource Allocation Procedure.

(1) **Determine the FPGA resource specification for a job:** based on user application requirements.
(2) **Select and reserve a pool of hardware resources:** to run a user job, based on the currently available resources.
(3) **Control the life cycle of user jobs:** it starts by running pre-job tasks such as configuring control registers. Next, it activates the FPGA resources needed for a job, run the job, and upon completion, it can release the hardware resources.

The admission control strategy starts by checking if there are enough resources that are available to service a user request. If there are sufficient resources, then the set of VRs that minimize the communication latency between hardware workloads are selected. Otherwise, the *FPGA Management service* from another zone is invoked. The assumption is that the computing resources in an IaaS are hosted in multiple locations that are divided in zones [25]. If it is not possible to find a VR allocation that satisfies the user needs, then the *Resource Allocation* module will return an adequate notification to the cloud user. Before elaborating on the resource allocation procedure, we first present the structure of the *VR/FPGA* and the *CID/VR* Tables (see Figure 14).

Because we seek to provision VRs instead of FPGAs as discussed earlier, we need a data structure that allows identifying the FPGA from which a VR is provisioned. In addition, it is necessary to know if a VR is available or is currently assigned to a VM. The *VR/FPGA Table* responds to these needs. In this section, we consider an abstracted view of the NoC in which each VR is attached to a router component to enable flexible communication (Section 4.2.1 details the NoC architecture). In that regard, we uniquely identify each VR in the system using the combination of the FPGA from which it is provisioned (FPGA_ID), the router component it is attached to (ROUTER_ID), and a VR bit (VR_ID) as illustrated in Figure 14. Next, when a VR is assigned, we should be able to identify the designated VM. We use the *CID/VR Table* data structure for that purpose. It maps the VR identifiers (FPGA_ID, ROUTER_ID, and VR_ID) to the CID of the VM. However, the CID is an integer value represented over 32 bits. However, using a 32-bit value in the data packet to identify a VM could

Table 4. Notations

Name	Description
n	Index of the accelerator request.
m	Index of the VR.
x_{ij}	Binary decision variable that represents whether VR_j is assigned to the $accelerator_i$ ($x_{ij} = 1$) or not ($x_{ij} = 0$).
c_{jk}	Communication latency from VR_j to VR_k.
$link_{iq}$	Binary constant defining whether data flow from $accelerator_i$ to $accelerator_q$ from a cloud user
l_i	LUT requirement of the $accelerator_i$.
L_j	LUT capacity of the VR_j.
f_i	Flip-flop requirement of the $accelerator_i$.
F_j	Flip-flop capacity of VR_j.
b_i	BRAM requirement of the $accelerator_i$.
B_j	BRAM capacity of the VR_j.
u_i	URAM requirement of the $accelerator_i$.
U_j	URAM capacity of the VR_j.
d_i	DSP requirement of the $accelerator_i$.
D_j	DSP capacity of the VR_j.

significantly increase the amount of FPGA resource used by the NoC. Therefore, we replace the 32-bit CID with a 5-bit VM identifier (VM_ID). It uniquely points to a VM when combined with the VR identifiers. Overall, the current layout allows an *FPGA Management service* to control up to 2,048 VRs across 32 FPGA devices.

VR Allocation: From the perspective of the VMM, the FPGA resources are represented as a dataflow graph (DFG) in which each node represents a VR, and the edges denote the communication latency between the VRs regardless of the physical FPGA from which they are provisioned. This DFG represents the *Architecture Definition* in Figure 14 and is an input from the CSP. Each node also embeds the hardware properties of the VR it corresponds to. In this work, the hardware properties that we consider are the number LUTs, FFs, BRAMs, DSPs, and URAMs. The *Graph Builder* keeps an updated list of the available VRs and their weighted connection as another DFG that is called *Architecture Graph* (see Figure 14). The *Architecture Graph* is obtained by pruning out from the *Architecture Definition*, the nodes representing VRs that are not in the "Available" state (see section 4.2.3). To request FPGA resources, cloud users submit a *Request Graph* in the form of a directed DFG. Just as in the case of the *Architecture Graph*, each node represents a VR. The edges denote the direction of communications between VRs. Each node of the *Request Graph* also carries information about the LUT, FF, BRAM, DSP, and URAM requirements. The *Placement Tool* starts by formulating an optimization problem (see Figure 14). It involves building a data file that captures the *Architecture Graph* and the *Request Graph*. It then loads the data file (.dat) and the placement model (.lsp) into a solver. We do not propose a new algorithm to solve optimization problems, but leverage innovation in operational research by using existing tools.

The optimization problem that is presented to the *Solver* in the *Placement Tool* can be presented as a set of equations in the form of a Mixed Integer Quadratic Program. The notations used in the problem formulation are summarized in Table 4. The optimization problem is expressed as follows:

$$Min \sum_{i=1}^{n} \sum_{q=1}^{n} \sum_{j=1}^{m} \sum_{k=1}^{m} c_{jk} \times link_{iq} \times x_{ij} \times x_{qk}. \quad (1)$$

Basically, we seek to decrease the latency of data movement between communicating hardware applications that are programmed in different VRs and belong to the same VM. It is, however, subject to a few constraints among which:

- **A VR can host at most one accelerator.** For consistency, a VR can only provision a single accelerator at a time. This constraint is illustrated by the following equation:

$$\sum_{i=1}^{n} x_{ij} \leq 1, \forall j = 1, \ldots, m. \tag{2}$$

- **An accelerator is assigned to at most one VR.** This prevents allocating multiple VRs to a node in the *Request Graph*.

$$\sum_{j=1}^{m} x_{ij} \leq 1, \forall i = 1, \ldots, n. \tag{3}$$

- **A VR must satisfy accelerator requirements.** As discussed earlier, the hardware properties that we consider are the number of LUTs, FFs, BRAMs, DSPs, and URAMs. Each VR must therefore be able to accommodate the hardware needs of the accelerator it will implement. In order words:

$$\begin{cases} \sum_{i=1}^{n} l_i \times x_{ij} \leq L_j, \forall j = 1, \ldots, m \\ \sum_{i=1}^{n} f_i \times x_{ij} \leq F_j, \forall j = 1, \ldots, m \\ \sum_{i=1}^{n} b_i \times x_{ij} \leq B_j, \forall j = 1, \ldots, m \\ \sum_{i=1}^{n} d_i \times x_{ij} \leq D_j, \forall j = 1, \ldots, m \\ \sum_{i=1}^{n} u_i \times x_{ij} \leq U_j, \forall j = 1, \ldots, m \end{cases} . \tag{4}$$

- x_{ij} **is a binary decision variable.** It ensures that the value of the decision variable is either 1 or 0. We have the following equation:

$$x_{ij} \in \{0, 1\}, \forall i \in \{1, \ldots, n\}; \forall j \in \{1, \ldots, m\}. \tag{5}$$

VR De-allocation: Finally, to release a VR (type (iv) request), the *table controller* simply updates the status of the corresponding VR in the *VR/FPGA table*, which puts the FPGA region in the pool of available resources and the *Request Handler* revokes the VM read/write privileges.

4.4 Communication Layer

4.4.1 Background on VirtIO. KVM is a virtualization extension present in Linux releases since kernel version 2.6.20. It transforms a Linux system into an hypervisor of type-1 (bare-metal) and benefits from decades of innovation in Linux process scheduling, memory management, device drivers, and so on—to manage VMs [57]. VirtIO is an API that abstracts IO devices in paravirtualized architectures [59]. It is the default communication scheme between guest and host in KVM environments since kernel version 2.6.25 [16, 36]. In short, it exposes VMs to virtual devices that handle the data traffic with the host. The virtual devices are accessible through front-end *virtIO drivers* installed in the GOS. The steps of a write operation with VirtIO are illustrated in Figure 15 [42]. ❶ At first, an application in the GOS writes into the *VirtIO driver*. ❷ The *VirtIO driver* buffers the value to write in the virtual memory space. ❸ VirtIO then notifies KVM of the presence of some data to transfer. This step is necessary as writing in physical devices can only be made at privileged level in the host. Since, VirtIO abstracts the hardware, it is not aware of the physical device that

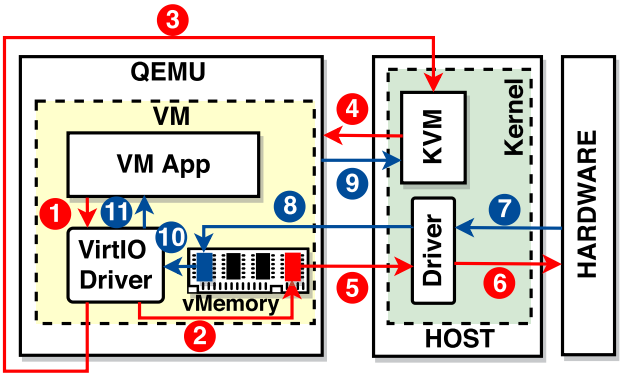

Fig. 15. Illustration of the IO Operation with VirtIO.

will be invoked. In consequence, it cannot issue the right system call at the host level. QEMU is therefore needed as a mediator, since it is a process in the host OS (HOS). ❹ Next, KVM issues a *vmexit* and transfer the control to QEMU. ❺ Since QEMU can access the whole virtual address space, it issues a system call to access the physical hardware in the host with the data that were previously buffered in step (2). ❻ The kernel in the host handles the write operation as access to the hardware requires kernel privileges. ❼ Upon completion, a flag value is returned to the hardware driver. ❽ The flag value is forwarded to QEMU, since it issued the system call. QEMU then stores the returned value into the virtual address space. ❾ QEMU notifies KVM of the completion of the emulation. KVM transfer the control back to the VM by issuing *vmresume*. ❿ The *VirtIO driver* can now read the status of the write operation. ⓫ The *VirtIO driver* notifies the application in the VM of the completion of the write operation. Though transitions to the host kernel are required to access the hardware because of ×86 ring of privileges (step 5 in Figure 15), the context switches with QEMU (steps 4 and 9 in Figure 15) incur high overhead penalty, resulting in degraded IO performance. The Vhost protocol was implemented as an improvement. It removes the QEMU emulation of IO operations by enabling shared memory between the *VirtIO driver* and a *Vhost backend driver* in the host. It introduces notifications (*eventfd*) allowing the *VirtIO driver* to notify the *Vhost backend driver* when transmitting data. It also handles interrupts (*irqfd*) that makes it possible for the *Vhost backend driver* to asynchronously notify the *VirtIO driver* when there is incoming traffic. The Vhost protocol is mainly used for network traffic and is implemented by the Vhost-net driver [42]. Our proposed approach is similar to Vhost and provides VM access to FPGAs through PCIe interfaces.

4.4.2 Our Communication Protocol. To create the illusion of having the FPGA directly attached to a VM, we seek to expose a PCIe-like drivers to VM applications. We first envisaged using VirtIO-pci. However, its transport protocol (VirtIO-serial [60]) presents some limitations: (1) serial ports only support one-to-one connections, (2) only one guest can communicate with the host at a time. Implementing a many-to-one communication will then require adding extra arbitration or opening multiple ports, and (3) serial ports only provides stream semantics. Supporting datagrams will necessitate an additional transformation layer. To implement efficient and flexible communication between guest and host, we therefore leverage the VirtIO-Vsock extension (present in Linux since kernel version 4.8) in place of VirtIO-pci [27, 41]. It is similar to Vhost in that it opens direct communication between guest and host without raising traps in the VMM. It implements sockets, which removes the need for guests to handle char devices as the frontend driver simply forwards

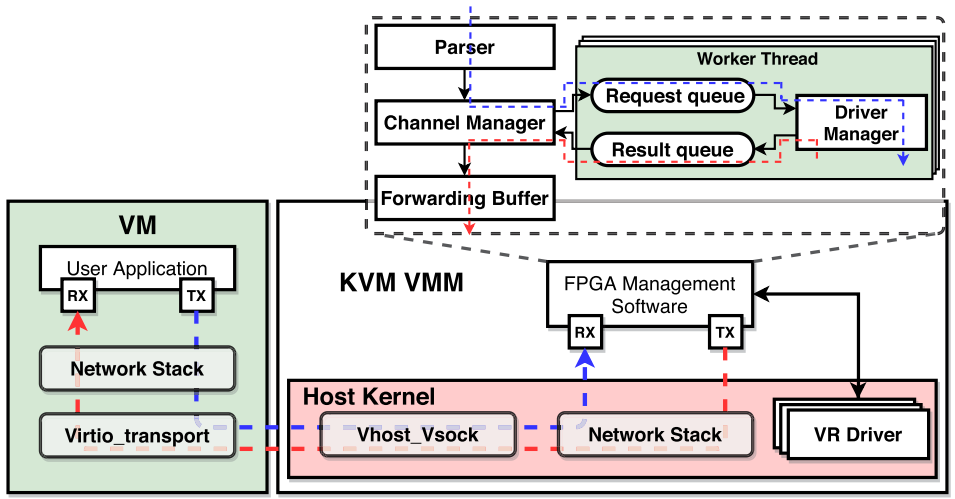

Fig. 16. Overview on the component involved in the communication architecture.

the traffic to the VMM. Its communication protocol is built upon the AF_VSOCK socket family added to Linux kernel from version 4.8. Just as in the case of Vhost, memory pages are shared between guest and host, which allows to bypass QEMU, and improve IO performance. It also supports many-to-one communication and allows both stream and datagram semantics. Finally, it uses the in-kernel network stack instead of going through the physical network, which is prone to connection shortages due to control features such as firewall configurations. One of the major changes introduced by this socket family resides in its addressing scheme. Addresses are made of a 32-bit CID combined with a 32-bit *port number*, as opposed to IP addresses and ports in TCP/IP and UDP sockets. The host always keeps a well-known CID while each guest is assigned a unique CID at boot time by the VMM. Our communication protocol defines two types of communication:

(1) **Requests for FPGA allocation/de-allocation:** these types of communication are destined to the VMM. Section 4.3 already elaborated on these types of request.
(2) **IO requests to FPGAs**: these correspond to read/write operations on FPGA accelerators.

IO requests to FPGAs: The proposed communication stack enables fast data movement between user applications and hardware accelerators on FPGA. At the host level, the *FPGA Management Software* interfaces with the hardware. It is responsible for reading or writing into specific VRs on FPGA. Figure 16 provides details on the communication components and the *FPGA Management Software*. Each IO operation (read/write) from the user applications in the guest generate and equivalent operation in the *FPGA Management Software* through remote procedure call. The VirtIO_transport stack is leveraged for efficient communication between GOS and HOS. The *FPGA Management Software* is mainly composed of a *parser*, a *channel manager*, *forwarding buffers*, a set of *request/result queues*, and *driver managers*. The *Parser* identifies the destination of each incoming traffic and forwards the packets to the appropriate communication channel. The *Channel Manager* controls the access to hardware by launching a new *"worker thread"* that services the communication between each VM-VR binding. The *Request* and *Result queues* implement isolated IO access by only allowing single read/write operation into each separate VR at a time. While the access to different VRs can be done in parallel as each *"worker thread"* only access a single VR address space, IO requests to the same VR on FPGA are serialized.

The *driver manager* in each *"worker thread"* is in charge of open, read, and write operations on each VR driver located in the */dev/* folder of the host. Finally, the *Forwarding Buffer* returns FPGA outputs to the VMs by writing into the buffers that are shared between guest and host.

5 EXPERIMENTAL RESULTS

5.1 Evaluation Set Up

We prototype the proposed architecture in a cloud configuration comprising a single server that runs the VMs and hosts an FPGA. The host server is a Dell R7415l EMC server with a 2.09 GHz AMD Epyc 7251 CPU and 64 GB of memory. The server runs on CentOS7 with a kernel of version 4.10.0. A Virtex UltraScale+ FPGA (xcvu9p-flga2104-2L-e) board serves as testing device. The FPGA is connected to the server through a PCIe Gen3 × 8 interface. We use Vivado 2019.1 to synthesize, place, and route the designs, as well as collecting design performance results (area overhead, power consumption, and maximum frequency). QEMU 2.11.50 emulates the VMs, each VM running on Ubuntu 16.04.01 with 4 GB of RAM.

5.2 Evaluation Methodology

The experiments on the proposed FPGA virtualization architecture will be organized in three major sections reflecting the virtualization layers described in Section 4:

(1) **VirtIO-Vsock Communication Cost:** First, we implement and measure the execution time of some image processing functions (on 3000 × 3000 input images) and a matrix operation (over 100 × 100 entries) under three execution environments: (1) In the host without FPGA acceleration, (2) FPGA support added to the host, and (3) VM with virtualized accelerators on FPGA. This study shows the minimal IO performance degradation incurred by the FPGA virtualization stack. Next, we evaluate the time it takes a VM to write and read (round trip) to/from the VR memory for various payload sizes. We also investigate the impact of increasing the number of VMs on the FPGA access time.
(2) **Soft-NoC Overhead and Performance:** In this section, we will study the area overhead of the VRs and routers, as well as the maximum frequency and power consumption outcome of some design choices. We will also highlight the performance benefits of enabling on-chip communication as opposed to relying on middleware data copy between the FPGA accelerators in a VM domain.
(3) **Case Study:** we provide an example of application design and deployment. The study shows a setup in which machine learning, computer vision, and cryptography hardware accelerators space-share a cloud FPGA.

Throughout the discussion on experimental results, we compare the achieved performances to recently published research.

5.3 Observations

5.3.1 VirtIO-Vsock Communication Cost. To measure the communication cost of the proposed architecture, we consider four VMs in two configurations: (1) 1 CPU and 4 GB of RAM and (2) 2 CPUs amd 4 GB of RAM. At first, we allocate a VR to each VM, and accelerate four applications (Sobel filter, matrix multiplication, Robert Cross, and smoothing) on FPGA. We specifically compare the execution times (executing the application and accessing the results) to assess the communication overhead introduced by the VirtIO-Vsock stack. Table 5 summarizes the findings. We note that the FPGA acceleration allows the VMs and Host to outperform the native execution (up to ~17× faster than native). However, the additional communication layer that transports data

Table 5. Execution Time Comparison

Applications	VM + VR	Host + FPGA	Host (without FPGA Acceleration)
Sobel	251.7 ms	251.68 ms	891.71 ms
Matrix Multiplication	4.05 ms	4.02 ms	70 ms
Robert Cross	107.91 ms	107.88 ms	818.29 ms
Smoothing	359.56 ms	359.54 ms	837.94 ms

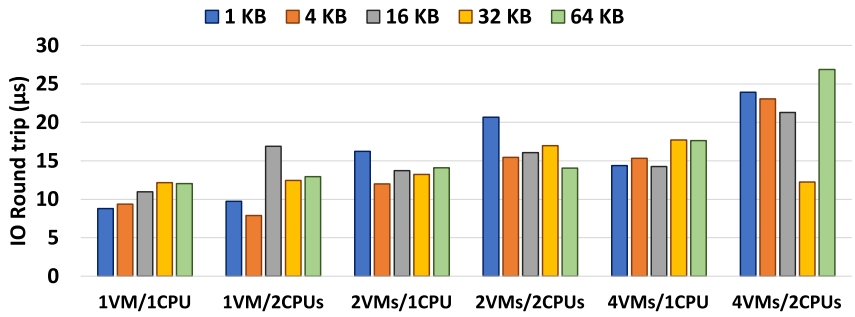

Fig. 17. Cost of IO trip through VirtIO-Vsock.

from the VMM to VMs introduces a data movement penalty. For instance, the overall execution time of the matrix multiplication when run from the host with FPGA acceleration is 4.02 ms, which is about 30 μs faster running the same job from a VM. Overall, the speed up mostly depends on the performance of the hardware accelerators, since the VirtIO-Vsock architecture does not introduce a significant communication overhead.

After comparing the execution times of native and FPGA-accelerated configurations, we study the round trip times when scaling the number of VMs and the size of the packets. Figure 17 presents the recorded results. The first observation is that the FPGA access time tend to increase with the number of resources allocated to the VMs regardless of the size of the packets. For instance with 1VM and 1CPU, it takes ~8 μs to complete the IO round trip. With a VM running on 2CPUs, it takes about ~10 μs. This is imputable to the overhead introduced by QEMU that emulates virtual CPUs in the VMs in addition to the scheduling taking place in the HOS: Assigning more resources to VMs tends to slow down the host server. We also note that the round trip times oscillate between 8 and 26 μs, which is smaller than the average 600 μs in Reference [12] (about 35× slower), and is far from the minimum minute needed to provision FPGAs in Reference [64]. Reference [14] reported about 15 μs for one VM to perform a round trip with a payload of size 4 KB, which is about twice greater than the 8 μs we obtain with one VM for the same payload size. This difference is can be justified by the VirtIO-Vsock efficient communication scheme that avoids context switches between VM and VMM address spaces.

5.3.2 Soft-NoC Overhead and Performance. We start by looking at the area overhead and latency of the VRs. Next, we study the resource and power consumption of the routers. Finally, we examine the operating frequency of the routers and the benefits of enabling on-chip communication.

VR Area and Latency: the results reported do not consider the *USER REGION* (see Figure 11) embedded within each VR as its area and latency vary with the CSP configuration and user design. Therefore, we focus on studying the static components that cannot be modified by the users. Table 6 shows the resource utilization recorded for data widths ranging from 32 to 256 bits. For each component of a VR, the table shows the number LUTs, FFs, and LUTRAMs that are utilized.

Table 6. Resource Utilization of the VRs

Data Width (bits)	AVC			FIFO			Validity Checker			Other		
	LUT	FF	LUTRAM	LUT	FF	LUTRAM	LUT	FF	LUTRAM	LUT	FF	LUTRAM
32	0	5	8	25	39	24	21	19	0	209	19	0
64	0	5	8	24	71	40	21	19	0	417	19	0
128	0	5	8	25	135	80	19	19	0	833	19	0
256	0	5	8	26	263	152	19	19	0	1665	19	0

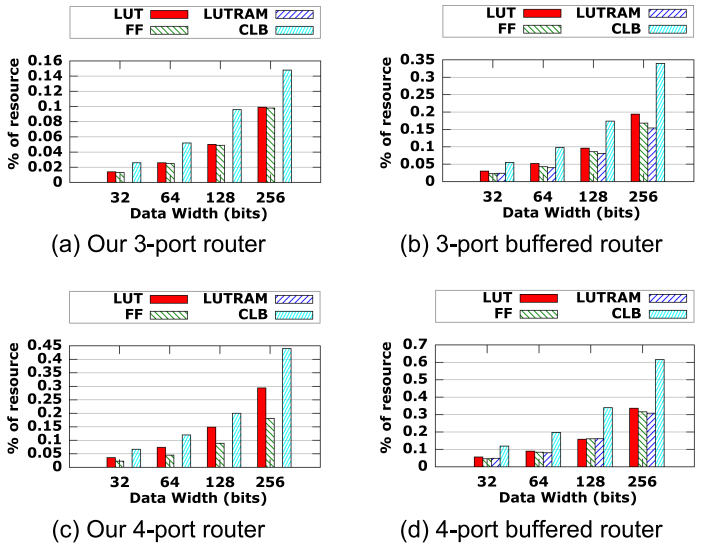

Fig. 18. Resource overhead of the routers.

As first observation, the area overhead of the AVC and *Validity Checker* remain essentially constant regardless of the data width. This is because these components essentially process the VM_ID that has a fixed size. However, the size of the FIFO and the other components such as the muxes at the input and output interfaces grows with the data width as these components process entire packets. Overall, the area overhead of the static components in the VRs remains minimal (~0.15% of LUTs). We also recorded a maximum frequency of operation ranging from 980 MHz for 32 bits data width down to 825 MHz on 256 bits data width. The time needed to stream a packet from the VR interface to the *USER REGION* mainly depends on the depth of the AVC as each packet needs to be checked against the content of the AVC. For instance, if the AVC has a single entry, then it takes 11 cycles to decide whether the packet will be accepted or rejected. With an AVC of 16 entries, it will take at most 26 cycles to decide. As a summary, it takes 4 cycles for the *Access Monitor* to pre-process a packet, 3 cycles to the *Validity Checker* to pull the packet from the input queue and extract the header, n cycles to compare the VM_ID to each entry in the AVC (where n represents the number of entries in the AVC), 2 cycles for the *Validity Checker* to decide whether to accept or reject a packet, and 1 cycle to the *Access Monitor* to return in a state where it can receive new incoming traffic. Finally, it takes 1 clock cycle to forward packets out of the VR.

Router Evaluation: In Figure 18, we study the resource utilization of the routers. We observe that reducing the number of ports significantly reduces the hardware footprint of the routers. In fact, the three-port routers use about 55% fewer registers and save about 35% of LUT logic compared to their four-port counterpart (see Figures 18(a) and 18(c)). Traditional routers

Table 7. Bandwidth and F_{max} Comparison with Previous Work

32-bit Router	#LUT	#Wire	Fmax (MHz)	BW/LUT (Gbps)	BW/Wire (Gbps)
CONNECT [53]	593	43	313	0.034 (×7.76↓)	0.233 (×5.56↓)
LB_Flex [40]	654	36	640	0.063 (×4.19↓)	0.569 (×2.28↓)
Hoplite [34]	60	36	638	0.340 (×1.29↑)	0.567 (×2.28↓)
LB_Fast [40]	76	36	1001	0.843 (×3.19↑)	0.890 (×1.46↓)
Our 3-port	167	34	1377	0.264	1.296
Our 4-port	437	34	1122	0.082	1.056

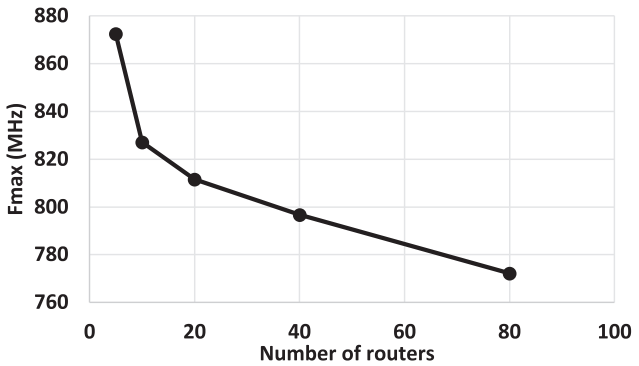

Fig. 19. Maximum frequency of the NoC.

that buffers data even show a higher resource utilization with additional LUTs, registers, and LUTRAMs (see Figures 18(b) and 18(c)).

In addition to the area, optimizing the routers also results in a higher operating frequency. Table 7 compares the maximum frequency and bandwidth of our routers with recent research. We observe that our routers have a higher operation frequency than the routers of LinkBlaze [40], Hoplite [34], and CONNECT [53]. The 32-bit routers in CONNECT and Hoplite achieved 313 MHz and 638 MHz on UltraScale+ devices [40]. This is far from the 1.3 and 1.1 GHz that is achieved respectively by our three-port and four-port routers on a similar device. Further, we compare bandwidth results for 32-bit routers to previous work. Our three-port router has 5.56× better bandwidth per wire than CONNECT, 2.28× better than Hoplite and LinkBlaze Flex, and 1.46× better than LinkBlaze Fast. Similar observations can be made for the four-port router. The bandwidth per LUT, however, draws a different picture. Hoplite and LinkBlaze Fast perform better than our routers, as they use about 2.7× fewer LUTs. This is due to the fact that they are less flexible. Hoplite implements a lightweight deflection and is unidirectional, which drastically reduces the size of the routing logic. LinkBlaze Fast routers only have three ports (two inputs and one output), resulting in lower LUT count. Though our routers can individually operate at ∼1 GHz, combining the routers into the NoC introduces delays of the datapaths. Figure 19 reports the maximum frequency of our proposed NoC when we increase the number of routers. Here, we only report the timing analysis results on the clock domain of the routers. We do not report timing results including the VRs as clock domain crossing is properly handled using queues. Between 5 and 80 routers (corresponding to 10 and 160 VRs), the maximum frequency drops from 872 MHz down to 772 MHz. Though it may not be practical to divide a device into as high as 160 VRs, this result shows that even for

Table 8. Comparison of Communication Performances to the Baseline

# of routers (32 bits)	# of VR	Best scenario			Worst scenario		
		Comm. time (ns)	Comparison to host baseline	Comparison to VR baseline	Comm. time (ns)	Comparison to host baseline	Comparison to VR baseline
1	**2**	1.378	×7256 ↑	×14513 ↑	11.024	×907 ↑	×1814 ↑
5	**10**	9.59	×1046 ↑	×2092 ↑	99.736	×100 ↑	×200 ↑
10	**20**	22.04	×453 ↑	×907 ↑	246.848	×40 ↑	×80 ↑
20	**40**	42.96	×233 ↑	×465 ↑	498.336	×20 ↑	×40 ↑
40	**80**	93.28	×107 ↑	×214 ↑	1100.704	×9 ↑	×18↑
80	**160**	182.08	×54 ↑	×109 ↑	2166.752	×4 ↑	×9 ↑

large configurations, our proposed NoC can still route packets close to the maximum frequency of UltraScale+ devices (~900 MHz [73]). Overall, each router can forward packets in 2 clock cycles, and direct links between VRs enable 1 clock cycle communication latency, allowing to reach a bandwidth of 28 Gbps on 32-bit routers.

Benefits of On-chip Communication: One may question on the necessity of implementing a soft-NoC to support hardware elasticity. In fact, designing FPGA accelerators is a time-consuming process depending on the complexity of the design to implement. Considering a context in which a user already programmed specific functions in a cloud FPGA, leveraging the deployed accelerators instead of redesigning an entire hardware stack is beneficial in term of productivity. It could also be more cost-effective as a hardware accelerator could be re-used by several hardware workloads in a VM domain. In addition, vendor tools such as Vivado often produce high-performance results for small modules of a design [38, 44]. Thus, dividing large accelerators into smaller modules that are connected through low-latency communication channels could result more efficient resource sharing with reduced data movement overhead. Another advantage is the low communication overhead compared to relying on software functions to initiate data movement between hardware accelerators. Considering that VM round trips (respectively, host round trips) to the FPGA could take in average ~20 μs (respectively, ~10 μs), we compare hardware-level data copy with that of software. We scale the number of virtual regions on an FPGA from 2 to 160 (which respectively corresponds to having 1 to 80 routers in the on-chip interconnect) and record the time required for transferring a packet between the two most distant VRs in term of routing hops. We consider two operating conditions: (1) The best scenario: There is no congestion. (2) The worst scenario: The on-chip interconnect is congested. Each router on the way is overloaded, which introduces routing delays. Table 8 summarizes the experimental observations. In the best scenario and with the FPGA divided into 160 regions, transferring a packet takes 182.08 ns, which is 54× (respectively, 109×) faster than an equivalent operation by the host (respectively, the VM). In the worst scenario, with the same quantity of VRs in an FPGA, the communication uses ~2 μs, which is 4× (respectively, 9×) faster than an equivalent operation by the host (respectively, the VM). This gap linearly increases as the number of VRs hosted on the FPGA decreases. Overall, this teaches two major lessons: (1) *implementing on-chip communication* support between the VRs drastically improves the throughput compared to letting a VM or the host copy the data between the accelerators on a chip. It also achieves similar performance than traditional FPGA virtualization architectures (see Figure 4(b)) for independent accelerators. (2) *Achieving higher throughput* is tightly associated to decreasing the number of regions provisioned on a single FPGA. Increasing the number of VRs on an FPGA increases the communication overhead. Depending on the performance and hardware utilization

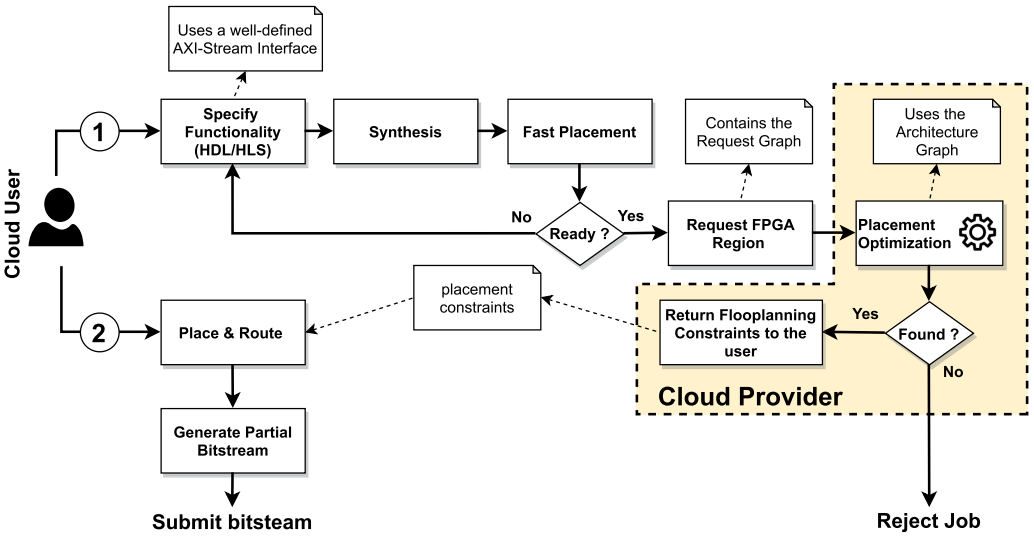

Fig. 20. Compilation and deployment steps.

tradeoff that an IaaS targets, cloud architects can divide the FPGAs into an adequate number of VRs.

5.4 Example of Application Design and Deployment

In this section, we describe the design flow and show an example of applications co-hosted on a multi-tenant cloud FPGA.

5.4.1 Design and Deployment Flow. We use the compilation and deployment steps illustrated in Figure 20. Initially, the cloud user designs the functionality using the "*Out-Of-Context (OOC)* Design Flow" [74]. The OOC flow allows fast synthesizing and placement of stand-alone modules and produces OOC design checkpoints. This provides a good estimation of the hardware resources needed for a workload. Once ready to program the actual hardware, the cloud user can build the *Request Graph* (see Figure 14) and request one or multiple VRs to the CSP. If the *Placement Tool* finds a satisfactory VR assignment, then it returns the adequate floorplanning constraints to the cloud user. The user can then place and route the hardware designs "In-Context" and submit partial bitstreams to the CSP. The runtime IO operations to the hardware accelerators are carried out through the communication layer described in Section 4.4 and evaluated in Section 5.3.1.

5.4.2 Case Study: FPGA space-sharing in the cloud. As we mentioned previously, the CSP is responsible for dividing an FPGA into a desired number of VRs. In this example, we split the FPGA into five different VRs of different sizes. Figure 21 shows the FPGA layout with the resource utilization of each dedicated area. The user designs can only be hosted in the VRs. We use Vendor IPs to implement the PCIe interface in the "*PCI Block*." The "*UNASSIGNED*" region is used to place and route the soft-NoC. Since the microbenchmarks used in Table 5 only use about 1.5% of the chip area, we use larger workloads for better illustration. The FPGA is shared between two VMs. The first one executes two independent functions: encryption/decryption with 128-bit AES and edge detection using the Canny algorithm. The second VM accelerates a *convolutional neural network (CNN)* on FPGA. It implements the Split-CNN architecture described in Reference [10], that partitions the input images into small patches that are processed independently. The architecture has

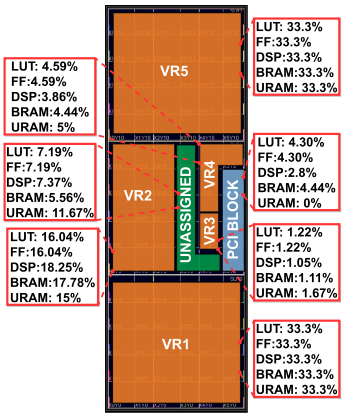

Fig. 21. FPGA layout.

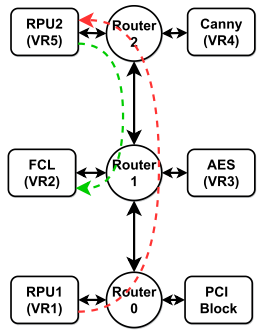

Fig. 22. Soft-NoC layout.

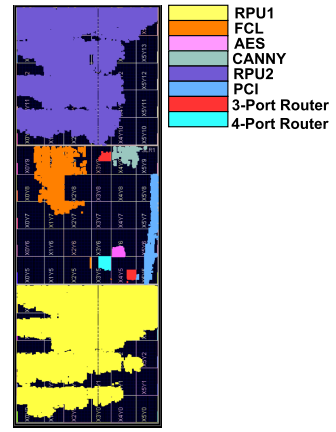

Fig. 23. Placement on FPGA.

three layers: two *region processing units (RPU)* that execute the convolutions, and a *fully connected layer (FCL)*. Given that the FPGA provisions five VRs, the soft-NoC only needs three routers: one four-port router and two three-port routers as illustrated in Figure 22. Based on the resource need of the applications, we obtain the VR allocation shown in Figure 22. For instance, the Split-CNN needs 401543 LUTs, 268270 FFs, 26.5 BRAMs, and 188 DSPs, which is not provisioned in any of the VRs. Therefore, we divide the design into three sub-modules RPU1, RPU2, and FCL that are programmed on VR1, VR5, and VR2. Figure 23 depicts the placement result on FPGA. RPU1 and RPU2 consume ~49% of VR1 and VR5; the FCL uses ~7% of VR2; AES covers ~10% of VR3; and Canny is spread over ~5% of VR4. Data flow through RPU1, RPU2, and FCL. The soft-NoC incurs a maximum of 46 clock cycles overhead (24 cycles to move data through RPU1 → RPU2 → FCL, and 11 cycles to validate packet at the interface of VR5 and VR2) in the data movement across the Split-CNN components, which remains in the order of a few nanoseconds. This is a considerable performance gain compared to the minimum 20 μs needed to move data between accelerators from VM software and 10 μs from the VMM. The combined three routers (respectively, the PCIe logic) use ~0.24% (respectively, ~0.29%) of the entire FPGA area.

6 CONCLUSION AND FUTURE RESEARCH

In this work, we proposed a hardware/software architecture for virtualizing FPGA resource in KVM clouds. The proposed architecture provides an FPGA abstraction to the users that consists in dividing the FPGA into "*virtual regions*." The architecture also features a communication layer that enables fast access to FPGA resources, and a model for efficient allocation of hardware in a way that minimizes the communication overhead at the hardware level. We presented a soft-Noc that supports hardware elasticity by enabling runtime allocation of additional hardware functions to a VM domain. Instead of solely relying on entire reconfiguration of user hardware to program additional functionality, it enables leveraging already deployed accelerators to serve in partial computations across the entire hardware domain of a VM. Experiments demonstrated the low resource utilization and high frequency of operation of our proposed hardware architecture. We also compared different results obtained with recently published research. However, provisioning multi-tenant FPGAs in the cloud incurs several challenges. First an foremost, space-sharing devices raises security concerns. Recent research has proven that attackers could leverage the reconfigurable nature of FPGAs to implement malicious circuits on space-shared devices. As a result, co-hosted hardware

designs could be the target of data leakage, IP theft, side-channel, fault-injection, covert-channel, and denial-of-service attacks [28, 33]. Furthermore, designing on a stand-alone FPGAs typically involves debugging and validation. Vendor logic analyzer such as ChipScope and SignalTap allow runtime monitoring of hardware design [35]. Other monitoring tools such as the System Monitor from Xilinx also allows tracking different device features such as the temperature and voltage levels [72]. These tools generally rely on dedicated interfaces such as the JTAG chain to collect runtime information. In a multi-tenant environment, additional features still need to be developed to ensure that each cloud user can only access information related the FPGA resources allocated to his virtual environment. Therefore, in future research, we intend to explore approaches to time-share FPGA debugging interfaces between multiple tenants. We also plan to study common countermeasures to the well-known security challenges of cloud FPGAs.

REFERENCES

[1] Amran A. Al-Aghbari and Muhammad E. S. Elrabaa. 2019. Cloud-based FPGA custom computing machines for streaming applications. *IEEE Access* 7 (2019), 38009–38019.

[2] Alibaba. 2020. Compute optimized instance families with FPGAs. Retrieved November 26, 2020 from https://www.alibabacloud.com/help/doc-detail/108504.htm.

[3] Amazon. 2019. Amazon EC2 F1 Instances. Retrieved November 26, 2020 from https://aws.amazon.com/ec2/instance-types/f1/.

[4] Amazon. 2019. Amazon EC2 Pricing. Retrieved November 26, 2020 from https://aws.amazon.com/ec2/pricing/on-demand/.

[5] Amazon. 2020. Amazon Web Services: Overview of Security Processes. Retrieved November 25, 2020 from https://docs.aws.amazon.com/whitepapers/latest/aws-overview-security-processes/hypervisor.html.

[6] Mikhail Asiatici, Nithin George, Kizheppatt Vipin, Suhaib A. Fahmy, and Paolo Ienne. 2017. Virtualized execution runtime for FPGA accelerators in the cloud. *IEEE Access* 5 (2017), 1900–1910.

[7] Anish Babu, M. J. Hareesh, John Paul Martin, Sijo Cherian, and Yedhu Sastri. 2014. System performance evaluation of para virtualization, container virtualization, and full virtualization using xen, openvz, and xenserver. In *Proceedings of the 4th International Conference on Advances in Computing and Communications*. IEEE, 247–250.

[8] Baidu. 2020. FPGA Cloud Server. Retrieved November 26, 2020 from https://cloud.baidu.com/product/fpga.html.

[9] Allen Jehle Bhavin Patel. 2020. Deploying AWS Storage Gateway on Linux KVM hypervisor. Retrieved November 26, 2020 from https://aws.amazon.com/blogs/storage/deploying-aws-storage-gateway-on-linux-kvm-hypervisor/.

[10] Pankaj Bhowmik, Md. Jubaer Hossain Pantho, Joel Mandebi Mbongue, and Christophe Bobda. [n.d.]. ESCA: Event-based split-CNN architecture with data-level parallelism on ultrascale+ FPGA. In *Proceedings of the IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 176–180.

[11] Bittware. 2020. How OVHcloud Uses FPGAs to Mitigate DDoS Attacks. Retrieved November 26, 2020 from https://www.bittware.com/resources/case-study-ovh/.

[12] Stuart Byma, J. Gregory Steffan, Hadi Bannazadeh, Alberto Leon Garcia, and Paul Chow. 2014. Fpgas in the cloud: Booting virtualized hardware accelerators with openstack. In *Proceedings of the 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'14)*. IEEE, 109–116.

[13] Sarath Chandra. 2019. Virtualisation at Alibaba Cloud. Retrieved November 26, 2020 from https://medium.com/@saisarathchandrap/virtualisation-at-alibaba-cloud-b20dea72efa1.

[14] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. 2014. Enabling FPGAs in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers*. ACM, 3.

[15] Spyros Chiotakis, Sébastien Pinneterre, and Michele Paolino. 2019. vFPGAmanager: A hardware-software framework for optimal FPGA resources exploitation in network function virtualization. In *Proceedings of the European Conference on Networks and Communications (EuCNC'19)*. IEEE, 47–51.

[16] Humble Devassy Chirammal, Prasad Mukhedkar, and Anil Vettathu. 2016. *Mastering KVM Virtualization*. Packt Publishing Ltd.

[17] A. Cortes, I. Velez, and A. Irizar. 2016. High level synthesis using vivado HLS for Zynq SoC: Image processing case studies. In *Proceedings of the Conference on Design of Circuits and Integrated Systems (DCIS'16)*. IEEE, 1–6.

[18] Michael Cui. 2020. Using Xilinx FPGA on VMware vSphere for High-throughput, Low-latency Machine Learning Inference. Retrieved November 26, 2020 from https://blogs.vmware.com/apps/2020/06/using-xilinx-fpga-on-vmware-vsphere-for-high-throughput-low-latency-machine-learning-inference.html.

[19] Guohao Dai, Yi Shan, Fei Chen, Yu Wang, Kun Wang, and Huazhong Yang. 2014. Online scheduling for fpga computation in the cloud. In *Proceedings of the International Conference on Field-Programmable Technology (FPT)*. IEEE, 330–333.

[20] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. 2012. High performance network virtualization with SR-IOV. *J. Parallel Distrib. Comput.* 72, 11 (2012), 1471–1480.

[21] Alibaba Cloud ECS. 2018. Deep Dive into Alibaba Cloud F3 FPGA as a Service Instances. Retrieved November 26, 2020 from https://www.alibabacloud.com/blog/deep-dive-into-alibaba-cloud-f3-fpga-as-a-service-instances_594057.

[22] Suhaib A. Fahmy, Kizheppatt Vipin, and Shanker Shreejith. 2015. Virtualized FPGA accelerators for efficient cloud computing. In *Proceedings of the IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom'15)*. IEEE, 430–435.

[23] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. 2004. *Reconstructing i/o*. Technical Report. University of Cambridge, Computer Laboratory.

[24] Silvia Gianelli. 2017. Baidu Deploys Xilinx FPGAs in New Public Cloud Acceleration Services. Retrieved November 26, 2020 from https://www.xilinx.com/news/press/2017/baidu-deploys-xilinx-fpgas-in-new-public-cloud-acceleration-services.html.

[25] Google. 2020. Regions and Zones. Retrieved November 29, 2020 from https://cloud.google.com/compute/docs/regions-zones.

[26] Tristan Groléat. 2020. Using FPGAs in an Agile Development Workflow. Retrieved November 26, 2020 from https://www.ovh.com/blog/using-fpgas-in-an-agile-development-workflow/.

[27] Stefan Hajnoczi. 2015. Virtio-vsock: Zero-configuration Host/Guest Communication. Retrieved November 26, 2020 from https://www.linux-kvm.org/page/KVM_Forum_2015.

[28] Festus Hategekimana, Joel Mandebi Mbongue, Md Jubaer Hossain Pantho, and Christophe Bobda. 2018. Secure hardware kernels execution in CPU+ FPGA heterogeneous cloud. In *Proceedings of the International Conference on Field-Programmable Technology (FPT'18)*. IEEE, 182–189.

[29] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture: A Quantitative Approach*. Elsevier.

[30] Huawei. 2020. FPGA Accelerated Cloud Server. Retrieved November 26, 2020 from https://www.huaweicloud.com/en-us/product/fcs.html.

[31] Huawei. 2020. FusionSphere OpenStack. Retrieved November 26, 2020 from https://e.huawei.com/en/cloud-computing/fusionsphere-openstack.

[32] IBM. 2020. Field Programmable Gate Arrays for the Cloud. Retrieved November 26, 2020 from https://www.zurich.ibm.com/cci/cloudFPGA/.

[33] Chenglu Jin, Vasudev Gohil, Ramesh Karri, and Jeyavijayan Rajendran. 2020. Security of cloud FPGAs: A survey. arXiv:2005.04867. Retrieved from https://arxiv.org/abs/2005.04867.

[34] N. Kapre and J. Gray. 2015. Hoplite: Building austere overlay NoCs for FPGAs. In *Proceedings of the 25th International Conference on Field Programmable Logic and Applications (FPL'15)*. 1–8. https://doi.org/10.1109/FPL.2015.7293956

[35] Günter Knittel, Stefanie Mayer, and Christian Rothländer. 2008. Integrating logic analyzer functionality into VHDL designs. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs*. IEEE, 127–132.

[36] KVM. 2020. Virtio Paravirtualized Drivers for kvm/Linux. Retrieved November 26, 2020 from https://www.linux-kvm.org/page/Virtio.

[37] Danielle Tchuinkou Kwadjo and Christophe Bobda. 2020. Late breaking results: Automated hardware generation of CNN models on FPGAs. In *Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC'20)*. IEEE, 1–2.

[38] Danielle Tchuinkou Kwadjo, Joel Mandebi Mbongue, and Christophe Bobda. 2021. Exploring a layer-based pre-implemented flow for mapping CNN on FPGA. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'21)*. IEEE, 116–123.

[39] Chris Lavin and Alireza Kaviani. 2018. Rapidwright: Enabling custom crafted implementations for fpgas. In *Proceedings of the IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'18)*. IEEE, 133–140.

[40] Pongstorn Maidee, Alireza Kaviani, and Kevin Zeng. 2017. LinkBlaze: Efficient global data movement for FPGAs. In *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig'17)*. IEEE, 1–8.

[41] Linux Programmer's Manual. 2020. vsock—Linux VSOCK Address Family. Retrieved November 26, 2020 from http://man7.org/linux/man-pages/man7/vsock.7.html.

[42] Eugenio Perez Martin. 2019. Deep Dive into Virtio-networking and vhost-net. Retrieved November 26, 2020 from https://www.redhat.com/en/blog/deep-dive-virtio-networking-and-vhost-net.

[43] Joel Mbongue, Festus Hategekimana, Danielle Tchuinkou Kwadjo, David Andrews, and Christophe Bobda. 2018. FPGAVirt: A novel virtualization framework for fpgas in the cloud. In *Proceedings of the IEEE 11th International Conference on Cloud Computing (CLOUD'18)*. IEEE, 862–865.

[44] Joel Mandebi Mbongue, Danielle Tchuinkou Kwadjo, and Christophe Bobda. 2019. Automatic generation of application-specific FPGA overlays with rapidwright. In *Proceedings of the International Conference on Field-Programmable Technology (ICFPT'19)*. IEEE, 303–306.

[45] Joel Mandebi Mbongue, Sujan Kumar Saha, and Christophe Bobda. 2021. Performance study of multi-tenant cloud FPGAs. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'21)*. IEEE, 168–171.

[46] Joel Mandebi Mbongue, Alex Shuping, Pankaj Bhowmik, and Christophe Bobda. 2020. Architecture support for FPGA multi-tenancy in the cloud. In *Proceedings of the IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP'20)*. IEEE, 125–132.

[47] Peter Mell, Tim Grance, et al. 2011. The NIST definition of cloud computing.

[48] Microsoft. 2010. Project Catapult. Retrieved November 26, 2020 from https://www.microsoft.com/en-us/research/project/project-catapult/.

[49] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. 2006. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technol. J.* 10, 3 (2006).

[50] Nimbix. 2020. Accelerate Your Intel FPGA Designs. Retrieved November 26, 2020 from https://www.nimbix.net/intel-fpga.

[51] Nimbix. 2020. Accelerate Your Workflows with Xilinx Alveo Accelerator Cards in the Cloud. Retrieved November 26, 2020 from https://www.nimbix.net/alveo.

[52] Leonhard Nobach, Benedikt Rudolph, and David Hausheer. 2017. Benefits of conditional FPGA provisioning for virtualized network functions. In *Proceedings of the International Conference on Networked Systems (NetSys'17)*. IEEE, 1–6.

[53] Michael K. Papamichael and James C. Hoe. 2012. CONNECT: Re-examining conventional wisdom for designing nocs in the context of FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 37–46.

[54] David Pellerin. 2016. Amazon EC2 F1 Instances. Retrieved November 26, 2020 from https://aws.amazon.com/ec2/instance-types/f1/.

[55] David Pellerin. 2017. FPGA Accelerated Computing Using AWS F1 Instances. Retrieved November 26, 2020 from https://www.slideshare.net/AmazonWebServices/fpga-accelerated-computing-using-amazon-ec2-f1-instances-cmp308-reinvent-2017.

[56] 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. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 13–24.

[57] RedHat. 2020. What Is KVM? Retrieved November 26, 2020 from https://www.redhat.com/en/topics/virtualization/what-is-KVM.

[58] Daniel Rozhko. 2018. *Memory and Network Interface Virtualization for Multi-tenant Reconfigurable Compute Devices*. Ph.D. Dissertation.

[59] Rusty Russell. 2008. virtio: Towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operat. Syst. Rev.* 42, 5 (2008), 95–103.

[60] Amit Shah. 2010. Features/VirtioSerial. Retrieved November 26, 2020 from https://fedoraproject.org/wiki/Features/VirtioSerial.

[61] Simon Sharwood. 2020. Baidu cloud catches up by offloading cloudy networking and storage to SmartNICs. Retrieved November 26, 2020 from https://www.theregister.com/2020/08/26/baidu_cloud_update/.

[62] Jim Smith and Ravi Nair. 2005. *Virtual Machines: Versatile Platforms for Systems and Processes*. Elsevier.

[63] TACC. 2020. Exploring alternate computer architectures. Retrieved November 26, 2020 from https://www.tacc.utexas.edu/systems/fabric.

[64] Naif Tarafdar, Nariman Eskandari, Thomas Lin, and Paul Chow. 2018. Designing for FPGAs in the cloud. *IEEE Des. Test* 35, 1 (2018), 23–29.

[65] Naif Tarafdar, Thomas Lin, Eric Fukuda, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. 2017. Enabling flexible network FPGA clusters in a heterogeneous cloud data center. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 237–246.

[66] Naif Tarafdar, Thomas Lin, Daniel Ly-Ma, Daniel Rozhko, Alberto Leon-Garcia, and Paul Chow. 2019. Building the infrastructure for deploying FPGAs in the cloud. In *Hardware Accelerators in Data Centers*. Springer, 9–33.

[67] Anuj Vaishnav, Khoa Dang Pham, Dirk Koch, and James Garside. 2018. Resource elastic virtualization for fpgas using opencl. In *Proceedings of the 28th International Conference on Field Programmable Logic and Applications (FPL'18)*. IEEE, 111–1117.

[68] Jagath Weerasinghe, Raphael Polig, Francois Abel, and Christoph Hagleitner. 2016. Network-attached FPGAs for data center applications. In *Proceedings of the International Conference on Field-Programmable Technology (FPT'16)*. IEEE, 36–43.

[69] Xilinx. 2018. Xilinx Launches the World's Fastest Data Center and AI Accelerator Cards. Retrieved November 26, 2020 from https://www.xilinx.com/news/press/2018/xilinx-launches-the-world-s-fastest-data-center-and-ai-accelerator-cards.html.

[70] Xilinx. 2019. UltraScale Architecture and Product Data Sheet: Overview. Retrieved November 26, 2020 from https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf.

[71] Xilinx. 2020. Plunify Enables InTime FPGA Development in the Cloud. Retrieved November 24, 2020 from https://www.globenewswire.com/news-release/2018/07/25/1542033/0/en/Plunify-Enables-InTime-FPGA-Development-in-the-Cloud.html.

[72] Xilinx. 2020. UltraScale Architecture System Monitor. Retrieved May 04, 2021 from https://www.xilinx.com/support/documentation/user_guides/ug580-ultrascale-sysmon.pdf.

[73] Xilinx. 2020. Virtex UltraScale+ FPGA Data Sheet:DC and AC Switching Characteristics. Retrieved November 29, 2020 from https://www.xilinxcom/support/documentation/data_sheets/ds923-virtex-ultrascale-plus.pdf.

[74] Xilinx. 2021. Vivado Design Suite User Guide Designing with IP. Retrieved July 26, 2021 from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug896-vivado-ip.pdf.

[75] Binbin Zhang, Xiaolin Wang, Rongfeng Lai, Liang Yang, Yingwei Luo, Xiaoming Li, and Zhenlin Wang. 2010. A survey on i/o virtualization and optimization. In *Proceedings of the 5th Annual ChinaGrid Conference (ChinaGrid'10)*. IEEE, 117–123.

[76] Fei Zhang, Guangming Liu, Xiaoming Fu, and Ramin Yahyapour. 2018. A survey on virtual machine migration: Challenges, techniques, and open issues. *IEEE Commun. Surv. Tutor.* 20, 2 (2018), 1206–1243.

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

[78] Ke Zhang, Yisong Chang, Mingyu Chen, Yungang Bao, and Zhiwei Xu. 2019. Computer Organization and Design Course with FPGA Cloud. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. ACM, 927–933.

Received January 2021; revised May 2021; accepted July 2021