Architecture Support for FPGA Multi-tenancy in the Cloud

Joel Mandebi Mbongue, Alex Shuping, Pankaj Bhowmik, Christophe Bobda
ECE Department, University of Florida, Gainesville FL, USA
Email: (jmandebimbongue, alexandershuping, pankajbhowmik)@ufl.edu, cbobda@ece.ufl.edu

***Abstract*—Cloud deployments now increasingly provision FPGA accelerators as part of virtual instances. While FPGAs are still essentially single-tenant, the growing demand for hardware acceleration will inevitably lead to the need for methods and architectures supporting FPGA multi-tenancy. In this paper, we propose an architecture supporting space-sharing of FPGA devices among multiple tenants in the cloud. The proposed architecture implements a network-on-chip (NoC) designed for fast data movement and low hardware footprint. Prototyping the proposed architecture on a Xilinx Virtex Ultrascale+ 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 (we achieved 6× higher FPGA utilization with our case study), which is one of the major goals of virtualization. Overall, our NoC interconnect achieved about 2× higher maximum frequency than the state-of-the-art and a bandwidth of 25.6 Gbps.**

***Index Terms*—Cloud, Field-Programmable Gate Array, Network-on-chip, Multi-tenancy, Elasticity**

I. Introduction

In recent years, Field-Programmable Gate Arrays (FPGAs) have increasingly been deployed in public cloud infrastructures provided by several technology companies such as Amazon and Alibaba [1], [2]. Developers can now take advantage of a rich library of pre-built hardware accelerators or implement custom features without purchasing FPGA boards, managing expensive licenses, setting up the operational infrastructure, and being obliged to work from a specific location. Though FPGAs in the cloud opens to a broader access to reconfigurable hardware, current commercial cloud systems have highlighted the lack of primitives and support allowing multiple workloads to space-share a single device. This could result in expensive utilization cost. An instance without FPGA can be about 8.5× cheaper than an equivalent with FPGA [3], [4]. Another issue is the waste of resource. In fact, FPGA devices most often gather more elements than what user workloads would typically need when considering the millions of components present in high-end FPGAs. As example, the Xilinx Virtex UltraScale+ FPGA deployed within Amazon F1 instances contains approximately 2.5 millions logic elements, 6800 DSP slices, and 75MB of BRAM [5].

Because the capacity of integration in FPGA technology continuously increases as some devices now achieve 9 millions of logic cells [6], we believe that single-tenant FPGA use in the cloud may soon be unsuited. It then becomes necessary to explore approaches to enable FPGA multi-tenancy. The National Institute of Standards and Technology (NIST) proposed several characteristics of cloud infrastructure among which is *resource pooling* and *rapid elasticity* [7]. The *resource pooling* refers to the consolidation of resources (storage, processing, memory, etc) to serve users in a multi-tenant model. On the other hand, the *rapid elasticity* consists in allowing the provision and release of resources. It also encompasses scaling services with the demand. Extending these concepts to cloud FPGAs could summarize in being able to run multiple accelerators on a single device simultaneously, and enable the allocation of additional FPGA resources at run-time.

In this paper, we propose an approach for FPGA virtualization in cloud infrastructure that addresses *resource pooling* and *rapid elasticity*. Since the elasticity assumes that resources can be acquired and ultimately released, we focus our study on FPGA sharing in the space domain. In order to allow logically isolated workloads to share a single device, we start by dividing FPGAs into disjoint regions. The regions are then interfaced to a network-on-chip (NoC) interconnect that allows extending the hardware domain of a task. Basically, a hardware task that is deployed over multiple FPGA regions can be seen as an application with several sub-functions that can communicate through the NoC, each one deployed in a separate location. Our contribution therefore includes:

1) Concept of elastic and multi-tenant FPGAs in the cloud.
2) A soft NoC for efficient on-chip communication between hardware accelerators. We optimize the NoC architecture considering the cloud needs, and provide a solution that can move data at about 1GHz for data width between 64 and 256 bits.
3) A case study on FPGA multi-tenancy and elasticity. It shows through a practical example the necessity of space-sharing, and illustrates the advantage of on-chip communication support for efficient elasticity.

In the rest of the paper, section II reviews recent research. Then, section III provides some background definitions. Next, section IV describes the components of the proposed soft NoC. Finally, section V shows some experimental results and section VI concludes the paper.

II. Related Work

A. FPGAs in the cloud

FPGA virtualization in the cloud has been discussed recently in several studies. Some contributions present solutions

to the temporal allocation of FPGA kernels [8]–[11]. They essentially explore techniques to successively allocate full or partial FPGAs to tenants over time. Other research illustrated architectures implementing spatial FPGA sharing by exposing FPGA regions labeled "*virtual FPGAs*" to cloud tenants. For instance, some architectures divide each physical FPGA into several locations that can be allocated to virtual instances (VI) [12]–[14]. Partial reconfiguration is then used for runtime update of VI's hardware kernels. Yet, the FPGA access remains mostly limited either by not allowing user custom designs but pre-built hardware functions, and/or not supporting direct on-chip communication. This restriction imposes middleware copy for data movement between accelerators. To minimize the data movement overhead, an on-chip interconnect can be used between virtualized hardware regions [15], [16]. Vaishnav et al. implement elasticity on cloud workloads by scheduling user jobs as they arrive [17]. Based on a list of waiting jobs and their needs, they use partial reconfiguration to repurpose the FPGA. Other work discuss security challenges of FPGA deployment in the cloud [18]. This aspect is out of the scope of this work.

B. Network-on-Chip

Network-on-chips have emerged as a solution to the lack of scalability of point-to-point links and buses. Several soft-NoCs implemented on FPGA have been proposed in the literature. In the FLexiTASK NoC, high-radix routers reduce the network diameter [19]. Schelle et al. explore NoC performance when modifying parameters such as the size of the network and the presence or not of virtual channels (VC) [20]. They showed for instance that VCs can lead to about $5\times$ increase in resource utilization, but allow higher throughput. They concluded that more logic should be spent on the NoC if the applications are more communication-oriented than compute. CONNECT is a flexible NoC generator for FPGA-based systems that allows the creation of arbitrary topologies [21]. Its flexibility however results in low F_{max} and high area overhead. Hoplite proposes a lightweight and bufferless router architecture that is capable of achieving high bandwidth for single-flit-oriented FPGA designs with low area overhead [22]. Our proposed NoC is inspired from Hoplite as we seek to minimize the hardware footprint of the NoC to make more resources available to user designs in the cloud. Maidee et al. present a topology that leverages under-utilized FPGA long wires at the edge of the device for fast data movement [23]. Our proposed topology similarly leverages long wires on FPGA devices. Discussions on hard-NoCs are out of the scope of this work.

III. FPGA Multi-tenancy and Elasticity

A. Background Concepts

The virtualization of computing components such as CPUs is well investigated and basically consists in running virtual CPU instructions on a physical processor. While instructions on a CPU can execute independently, FPGAs implement circuits that depend on the physical architecture of the underlying hardware. This therefore requires reserving some FPGA regions to place run-time workloads as opposed to scheduling instructions as in the case of CPUs. In the context of this work, we define **FPGA Multi-tenancy** as the capability of space-sharing the physical area of a device between hardware accelerators from different cloud users. The placement of several hardware kernels consequently imposes splitting the FPGA into non-overlapping areas that we call "*virtual region*" (VR). VRs represent the unit of virtualized FPGA resource in the cloud. We consider the **FPGA Elasticity** as a feature that enables assigning additional unit of FPGA virtualization to already deployed tasks run-time with support for on-chip sub-function communication.

B. Cloud Virtualization Model

In this work, we consider the virtual resource access flow illustrated in Figure 1. It starts with a user request to the cloud provider for setting up a virtual instance (VI). The user selects the resources to attach to the desired VI and can start running applications. Tasks can run as long as they do not violate the

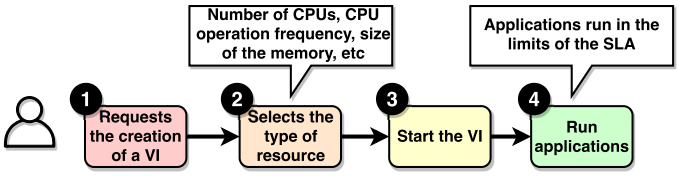
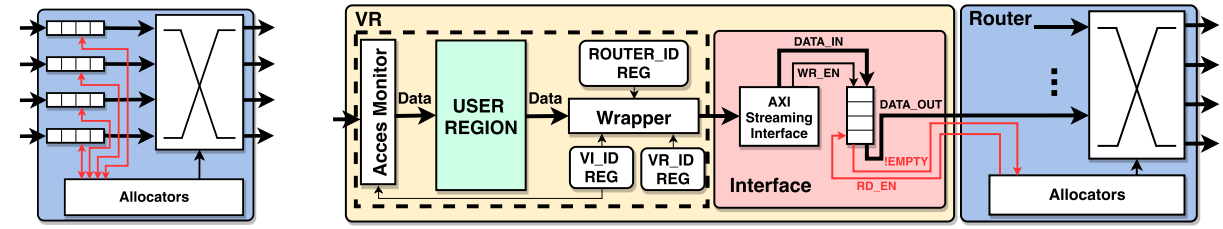


Fig. 1. Regular Virtual Instance Creation Flow

Service-Level Agreement (SLA). For instance, if a VI is set up with a disk of 1TB, it will not be possible to store more data until requesting additional storage. This flow is generally adopted in cloud infrastructures delivering VI. Our work seeks to enable selecting FPGA unit of virtualization as part of VIs. The size and shape of each VR is left to the cloud provider's choice just as they decide what unit of memory, storage, and processing they offer in their VI flavors. Since the amount of logic in an FPGA is finite, the same goes for the area of each VR. In consequence, the designs that are larger than a VR will be divided into modules by the user just as it would be the case if a design was bigger than an entire device. Next, the user will place a request for additional FPGA unit of virtualization. Because the two user regions will eventually exchange data, we propose to provide an efficient NoC interconnect as part of the Shell on FPGA. The NoC will also enable extending deployed workloads with additional functions at different VR. Our concept of elasticity differs from that of Vaishnav et al. [17], as we consider a model in which users fully control (run-time programming through partial reconfiguration) units of FPGA assigned to their domains by the cloud infrastructure.

IV. Proposed Network-On-Chip Architecture

In order to efficiently implement the *resource pooling*, we seek to maximize the number of concurrent workloads that can be deployed on a single device. In other words, we attempt to minimize the amount of resources consumed by the shell (NoC and IO controllers). We will however focus on optimizing the



(a) Buffered Bidirectional Router (b) Bufferless Bidirectional Router and VR Architecture

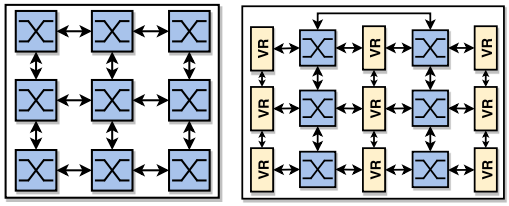
Fig. 2. Router optimization for F_{max} improvement and area reduction.

NoC architecture. Further, for fast data movement between VRs, the NoC should achieve device specification F_{max}, which is related to decreasing the number of LUTs on the datapaths.

A. Proposed Topology

While there are several topologies such as ring, star, hypercube, etc; we consider a 2D Mesh style for our NoC architecture. 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 attached to a PE). Figure 3a illustrates a general view of a 3×3 2D mesh. Mesh topologies have two defects in term 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 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 2 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 3b. It is a 3×3 mesh in which routers are connected to VRs. It implements a topology in which routers have at most 4 ports. As opposed to a regular mesh, each router is connected to 2 VRs, which decreases the hops. In order to keep the radix of 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 VRs, which allows offloading routers and streaming data every clock cycle between adjacent workloads.

Depending on the width of a device and the size of the VRs, the topology can be deployed in three different flavors: **(1) Single-Column:** in which the routers are lined up vertically on a few columns of configurable logic blocks (CLB). **(2) Double-Column:** it uses two columns of routers as in Figure 3b. In this mode, underutilized wires at the edge of the device are used to connect the two columns of routers. In fact, unless specified placement constraints, vendor tools tend to place and route designs closer to the center of the chip. By using wires at the edge, we take advantage of commonly wasted FPGA resources to provision additional VRs. **(3) Multi-Column:** it extends the previous mode with additional columns of routers and is suitable for wider devices.



(a) (b)

Fig. 3. (a) Traditional 2D Bidirectional Mesh Architecture. (b) Our proposed NoC topology. It reduces the radix of routers and enables direct inter-VR communication.

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 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 [24]. 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.

B. Router Component

1) 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 2a. The *Input Buffers* serve two purposes: (1) enabling minimized event of metastability between VR and router clock domains. (2) Temporary data storage when the destination is not ready. In order to forward traffic to the 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

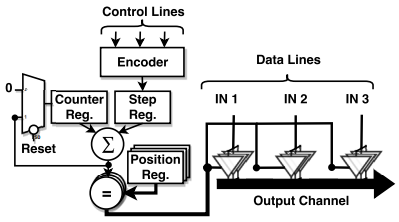


Fig. 4. Mutual Exclusion Logic

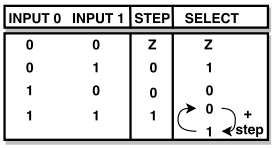


Fig. 5. 2-Input Encoder

inputs and m outputs, each of the output lines only needs $n-1$ switches since it is not the case that a VR will send data to itself. Each router therefore has $(n-1) \times m$ switches in the crossbar instead of $n \times m$. With 4-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 3b). This is simply a consequence of the absence of a fourth component to attach. Because one of the goals is to keep a low hardware footprint, we implement a 3-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 3 and 4 interfaces.

Kapre et al. observed that buffers can increase router resources by $20\% - 40\%$, which comes at the cost of area, delay, and power [22]. As in Hoplite, we therefore implement bufferless routers as illustrated in Figure 2b. We remove the buffers from the routers and keep data within VRs until the routers is ready 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 is 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 is 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 4 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, one packet is pulled from input interface at a time to establish fairness. Figure 5 shows the logic of the encoder.

To illustrate the management of mutual exclusion, consider a 4-port router with traffic coming from ports 1, 2 and 3 to port 4. Figure 6 summarizes how the *allocator* loads the packets. In cycle 1, there are incoming traffic from the 3 ports. The packets are routed one at a time. In cycle 4, when new data arrives at the 3 input ports, the data is loaded in the same way. From the third cycle, data will simply keep flowing out of the router because the inputs are pipelined.

2) Routing Procedure: In this section, we discuss the routing algorithm implemented in our NoC topology.

Although we opted for bufferless routers like Hoplite does, we do not implement deflection for two reasons. First, it may lead to unpredictable number of hops. Second, the routers

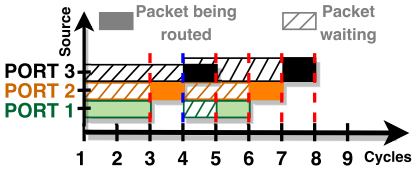


Fig. 6. Illustration of the mutual exclusion when packets at destination of Port4 of arrive simultaneously from Port1, Port2 and Port3 in a 4-port router.

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 routing decision is based on the content of each packet header. The packet structure is presented in Figure 7. The header has a fixed width of 16 bits and the payload as a configurable size. The header defines the destination of the packets. It is a combination of the VR_ID and ROUTER_ID. The VR_ID is represented on 1

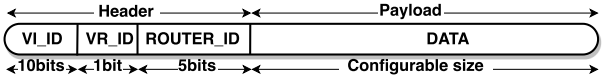


Fig. 7. Communication Packet Structure

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. The VI_ID uniquely identifies the VI to which the packet belongs. It is not actually used in the routing process, but at the VR interface to prevent sending packets to a VR belonging to a different VI. The ROUTER_ID occupies 5 bits and labels routers with integer values. The VI_ID occupies 10 bits, which allows handling up to 1024 VIs. Algorithm 1 summarizes the routing procedure.

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
```

The algorithm first checks the ROUTER_ID. If the current ROUTER_ID is greater (resp. smaller) than that of the packet being transmitted, the packet is pushed up (resp. pushed down). If the packet has reached the destination router, the VR_ID field is checked to determine the VR into which the packet will be injected.

In the next section, we discuss the structure of the VRs.

C. Virtual FPGA Region Architecture

The architecture of FPGA provisioned regions is illustrated in Figure 2b. The major component of the VRs is the *USER REGION*. It hosts the cloud user's custom designs and implements the partial reconfiguration paradigm. The VRs also feature an *Access Monitor* which only accepts packets from a specific VI. It removes the packet header and only forwards the payload to the *USER REGION*. The user designs only receive the payloads to prevent malicious application from trying to access resources out of a their domain. Developers are simply provided well-defined interfaces to implement in their design. Next, the cloud infrastructure selects the suitable VR that will host the hardware accelerator. Finally, it programs the design into the *USER REGION* inside the selected VR. At configuration time, the hypervisor edits the content of the VR registers. If the VR communicates with other FPGA regions, the router and VR identifiers of the destination are stored in the *ROUTER_ID* and *VR_ID* registers. The VI identifier is also written into the *VI_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. Details on algorithms implemented in the hypervisor to efficiently select the VRs to allocate to the VIs are out of the scope of this work.

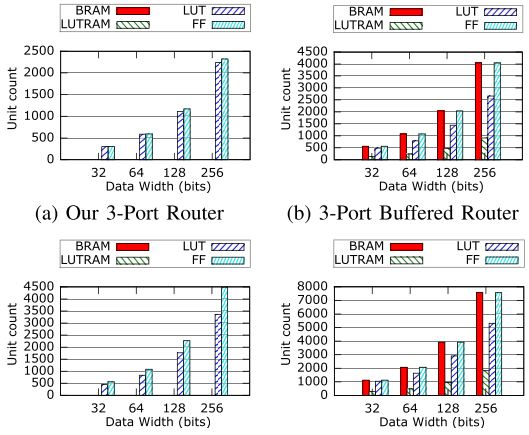
V. Experimental Evaluation

A. Evaluation Platform

We prototype the proposed architecture in a cloud configuration comprising two nodes. The first node runs the VIs on OpenStack Stein. It is an all-in-one deployment on a Dell R7415l EMC server running on a 2.09GHz AMD Epyc 7251 CPU with 64GB of memory. The second node hosts the FPGA. It is a Supermicro X10DAx servers with a 3.50GHz Intel Core i7-5930K CPU with 64GB of memory. Both nodes run CentOS-7 with a kernel of version 3.10.0. The servers are connected to a XR700 Nighthawk router operating at a bandwidth of 100Mbps. We use a Xilinx Virtex UltraScale+ FPGA or simply VU9P (xcvu9p-flgb2104-2-i) as testing device. Vivado 2018.2 is used to synthesize, place and route the designs.

B. Evaluation Methodolody

We will proceed in two steps. First, we evaluate the performance benefits of the optimizations discussed in section IV. We assess the performance of our NoC against some metrics such as area, power, maximum frequency, latency, and waiting time. We will also compare our proposed NoC to previous research. Next, we study an example case. We consider a cloud deployment in which multiple VIs are allocated some regions of the FPGA. We do not discuss the VR allocation flow as it is out of the scope of the work, but we analyze the outcome of sharing a device between several tenants by discussing FPGA access time and throughput. We want to demonstrate that sharing the FPGA outcomes in higher FPGA utilization and minimal loss in quality of service (QoS) compared to allocating a whole device to a single tenant. Next, we will compare our results to recent research on FPGA virtualization in the cloud.



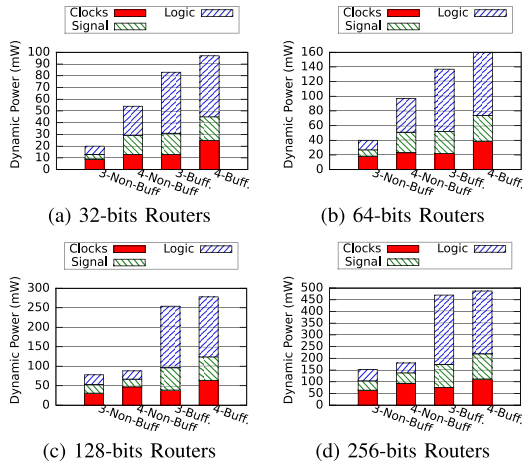
(a) Our 3-Port Router (b) 3-Port Buffered Router

(c) Our 4-Port Router (d) 4-Port Buffered Router

Fig. 8. FPGA Resource Utilization of the Router

C. NoC Evaluation

1) Resource and Power Consumption: In Figure 8, we study the resource utilization of the routers. It first evaluates the benefits of optimizing the number of interfaces of the routers. Next, it presents the advantages of removing buffers from the routers. Results are recorded for a data width ranging from 32 bits to 256 bits. The first observation is that reducing the number of port significantly reduces the hardware footprint of the router. In fact, Figure 8a and 8c show that 3-port routers uses about 40% less registers and save about 50% of LUT logic compared to the implementation with 4 interfaces. The router with buffers even show a more pronounced use of resources with additional LUTs, registers, BRAMs and LUTRAMs (see Figure 8b and 8d). The impact of router resources on power consumption is shown in Figure 9.



(a) 32-bits Routers (b) 64-bits Routers

(c) 128-bits Routers (d) 256-bits Routers

Fig. 9. Power Consumption Study of the Routers

First, the 4-port routers that are bufferless can consume up to 2.7× more power than their 3-port counterparts. Next, buffered routers consume up to 3.11× more power than bufferless implementations, the highest percentage being recorded from logic. These results demonstrate the benefits in area and power of optimizing the router architecture.

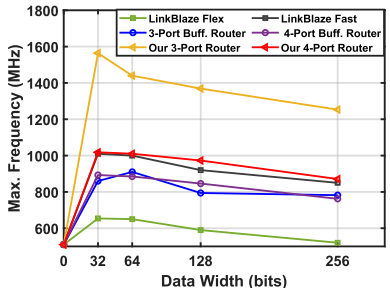


Fig. 10. Router Scalability Considering the Data Width

2) Maximun Frequency and Latency : In addition to the area and power benefits, the optimization of the router architecture also results in a higher operating frequency. In Figure 10, we compare the maximum frequency of various routers for data width between 32 and 256 bits. We compare our routers to the corresponding buffered implementations, as well as to LinkBlaze Flex and LinkBlaze Fast [23]. The maximum frequency tends to decrease when the data width increases. This is because larger data widths introduce more logic into the design, which results in additional delays on the data paths. We observe that our routers perform better than the buffered routers and the routers of LinkBlaze Fast/Flex. From the results reported in [23], CONNECT and Hoplite achieved 313MHz and 638MHz on a Virtex UltraScale+ FPGA. This is far from the 1.5GHz and 1GHz that is achieved respectively by our 3-port and 4-port routers on a similar device. Further, we compare bandwidth results for 32-bit routers to previous work (see Figure 11). Our 3-port router has 6.3× better bandwidth

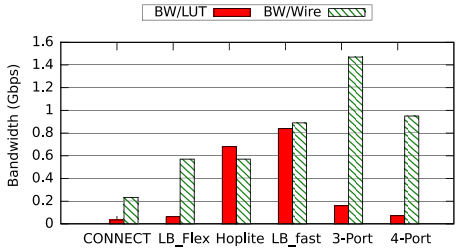
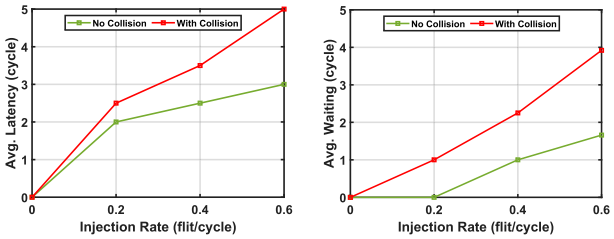


Fig. 11. Bandwidth Comparison to Previous Work

per wire than CONNECT, 2.57× better than Hoplite and LinkBlaze Flex; and 1.65× better than LinkBlaze Fast. Similar observations can be made for the 4-port router. The bandwidth per LUT nevertheless draws a different picture. Hoplite and LinkBlaze Fast perform better than our routers as they use about 5× less LUTs than our routers. 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 [22]. LinkBlaze Fast routers only have 3 ports (2 inputs and 1 output), resulting in lower LUT count [23].

We also evaluate our routers against various traffic patterns. Overall, an incoming flit needs two clock cycles to traverse a router. However, when the inputs are pipelined, only the first one will take two cycles. The following packets will be available at the outputs of the router after each cycle (see Figure 6). Figures 12a and 12b summarize the latency and



(a) (b)

Fig. 12. (a) Average latency study per injection rate. (b) Average waiting time study per injection rate.

waiting time observed on our 3-port router in two different configurations. First, we consider when flits arrive from all the interfaces with no collision. In other words, each output port of the router only receives traffic from one input port. With an injection rate of 0.6, the average latency observed is 3 clock cycles and the average waiting is 1.66 clock cycles. Next, we assess the latency and waiting time with collision. In this testing configuration, traffic from two ports target the third port of the router. We observe an increased latency compared to when there is no collision. It is just a consequence of having the packets waiting longer in the VR queues for their turn. In fact, Figure 12b shows a linear progression of the the waiting curve as the workload increases. The waiting time values when considering collision are about 2× higher than without collision, which reflects on the average latencies reported in Figure 12a.

D. Case Study: FPGA Multi-tenancy

1) FPGA division between tenants: To evaluate the FPGA multi-tenancy and elasticity when using our NoC, we consider 5 VIs (labeled VI1,...,VI5) deployed on the OpenStack cloud that access 6 VRs (labeled VR1,...,VR6) on FPGA. The assignment of VRs to VIs is as follows: VR1 is allocated to VI1; VR2 is allocated to VI2; VR3 and VR4 are allocated to VI3; VR5 is allocated to VI4; VR6 is allocated to VI5. For testing purposes we select 6 hardware accelerators from OpenCores [25]. The applications are: ***Huffman Decoder*** — that is typically used in streaming applications; ***FFT***—that is heavily used in signal processing; ***FPU***—it implements a single precision floating point unit; ***AES***—that is an encryption/decryption core over a 128-bit key. ***Canny Edge***—implements an edge detection algorithm. ***FIR***—is a commonly used filter in signal processing. Table I summarizes the VR allocation to VIs and the use of resources of test accelerators. VI3 initially implemented the FPU unit and later requested additional FPGA resource to implement encryption as the two could not fit into the area of VR3. To show the benefits of elasticity with on-chip communication, the FPU streams its output results directly to the AES encryption module through the NoC interconnect. The on-chip communication offers a bandwidth of 25.6 Gbps. Without communication support on the chip, moving data between two VRs will require middleware intervention to copy the data. This could cost around 50μs (Figure 14 reported a minimum of 28μs for directIO access), which represents a significant performance loss compared to the bandwidth of the NoC. On-chip communication support

is therefore of paramount importance to implement efficient hardware elasticity. For experimental purposes, we implement

TABLE I
VR ALLOCATION AND RESOURCE UTILIZATION OF THE APPLICATIONS

	LUT	LUTRAM	FF	DSP	BRAM
Huffman (VR1→VI1)	1288	408	391	0	1
FFT (VR2→VI2)	3533	92	4818	4	3
FPU (VR3→VI3)	4122	0	582	2	0
AES (VR4→VI3)	1272	0	500	0	0
Canny Edge (VR5→VI4)	2558	20	3825	0	18
FIR (VR6→VI5)	270	0	347	4	4

the *single-column* division of the FPGA (see section IV-A). Since we have 6 VRs, we will only need 3 routers (two 3-port routers and one 4-port router). The routers support 32-bits datapaths. Figure 13 shows a screenshot of area occupied by the NoC and each of the applications. For the sake of brevity, we do not show the rest of the shell that controls IO interfaces for off-chip communication. Because of the high capacity of integration of high-end FPGA architectures such as the UltraScale+ family, the size of each VR could easily be close to that of an entire legacy FPGA. For instance, the pblock defining VR5 occupies 1121 CLBs, or 8968 LUTs (0.22% of the LUTs in VU9P) which represents about 20% of some FPGAs from the 7-series [26]. This means that while a device from the 7-series may only be able to host about 5 instances of size equal to VR5, a VU9P device could deploy about 455 instances of those. This observation highlights the

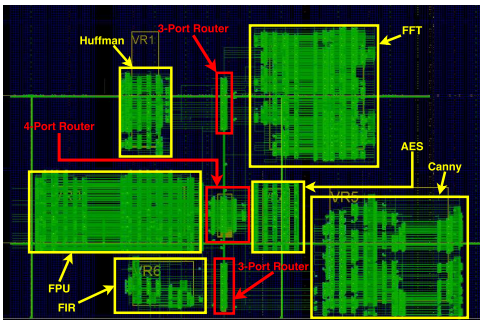


Fig. 13. Placement of the 6 Jobs from 5 VIs on a single device

need for spatial sharing support to increase device utilization. The NoC and applications illustrated in Figure 13 only used 1.71% of the CLB area of the FPGA, leaving enough room for additional workloads from cloud users. The 3-port and 4-port routers respectively cover 305 LUTs (0.03 % of the FPGA) and 491 LUTs (0.04% of the chip).

2) IO Trip and Throughput Study: first, we measure the overhead introduced by the cloud management software on the FPGA access time. We want to compare the IO performance in multi-tenant and single-tenant deployments to show that the spatial sharing of FPGAs does not significantly affect the QoS. We then consider two modes: **(1) Multi-tenant (Our approach):** all the 6 applications are deployed as illustrated in Figure 13. The VIs continuously write, then read from the accelerators and we record the IO trip times. **(2) Single-tenant (DirectIO):** The entire FPGA is successively allocated to each VI that runs write, then read operations and we record IO trip times. Figure 14 summarizes the average IO trip recorded time. It is observed that there is no significant difference in IO cost between the two schemes as they both simply consist in accessing FPGA registers from the host/guest operating systems. An IO access time penalty is however recorded when requests arrive simultaneously from different tenants at the entry point of the shared device. Such requests are queued in the cloud management software and the IO access delays observed are only in the order of a few microseconds. As example, an IO round trip to the AES core takes in average $31\mu s$ in the multi-tenant deployment while using about $29\mu s$ in the single-tenant FPGA allocation. On the other hand, accessing the FIR IP took in both cases an average of $31\mu s$. There are also cases where the IO requests performed better in the multi-tenant configuration. This means that we have achieved a 6× higher FPGA utilization rate as a single device is transparently running 6 different workloads. It is worth to note that these results were recorded in a configuration in which the FPGA was connected to the same physical server running the VIs (the FPGA node was purposely merged with the all-in-one OpenStack node for fair comparison with the directIO scheme). As we will discuss later, remotely accessing the FPGA incurs network transmission overhead.

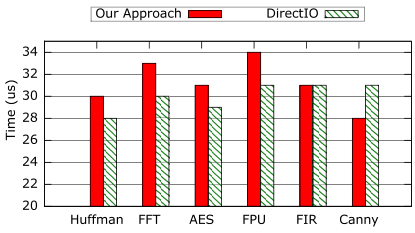
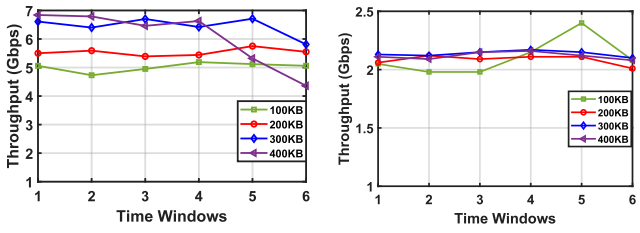


Fig. 14. IO Trip Comparison

We also study the throughput achieved on the multi-tenant cloud FPGAs. We continuously stream packets of size ranging from 100KB to 400KB between the VIs and hardware accelerators on FPGA, and record the average of throughput observed. Throughput data is collected over an hour of operation after 6 random time windows with all the VIs deployed on the server hosting the FPGA (Figure 15a) and with VIs remotely accessing the FPGA node over the Ethernet (Figure 15b).



(a) (b)

Fig. 15. (a) Throughput Study when the Virtual Instances are deployed on the physical Server hosting the FPGA devices. (b) Throughput Study when the Virtual Instances access FPGAs remotely.

When VIs access FPGA accelerators hosted on the same physical server, we observe a throughput reaching 7Gbps for 400KB payloads, which is about 2× higher than the software to hardware and hardware to software throughput reported in [27]. Up to 3× performance lost is however observed in distant FPGA access as the throughput is limited by the bandwidth of the Ethernet router (see section V-A).

3) Comparison with previous work: Table II puts into perspective our proposed architecture compared to other reported

FPGA enabled cloud schemes. As the table shows, DirectIO provides better performance compared to our approach but does not offer actual virtualization benefits such as resource re-allocation at runtime. Our approach appears as the best trade-off as it enables runtime re-allocation, hardware elasticity, local communication between VRs hosted on the same device. The work presented in [15] has a lower IO trip time, but is technology-specific as it only works for KVM clouds.

TABLE II
CLOUD FPGA ARCHITECTURE COMPARISON

Works	Runtime Re-allocation Support	Hardware Elasticity Support	On-Chip Com. Support	IO Trip Cost (in μs)
DirectIO	No	Yes	Yes	28
Our Work	**Yes**	**Yes**	**Yes**	**30**
[12]	Yes	No	No	15
[13]	Yes	No	No	600
[15]	Yes	Yes	Yes	26
[17]	Yes	Yes	No	–
[28]	Yes	No	No	8000
[29]	Yes	No	No	16000

VI. CONCLUSION

This work proposed an approach to enable spatial sharing of FPGA resource between multiple tenants in the cloud. We leverage a NoC architecture to implement elasticity. In the context of this work, we considered the elasticity as the ability to assign additional FPGA components to users at runtime. The proposed NoC makes it possible assign multiple FPGA regions to users and implement fast data movement to support on-chip communication between running workloads. Experiments demonstrated the low resource utilization and high frequency of operation of our architecture, as well as an increased FPGA utilization.

ACKNOWLEDGEMENT

This work was partially supported by the ONR under the Grant CCN 0402-17643-21-0000, and the Air Force Research Lab AFRL/RIGA Cyber Assurance Branch, Rome NY.

REFERENCES

[1] D. Pellerin, "Amazon ec2 f1 instances," https://aws.amazon.com/ec2/instance-types/f1/, 2016.

[2] A. C. ECS, "Deep dive into alibaba cloud f3 fpga as a service instances," https://www.alibabacloud.com/blog/deep-dive-into-alibaba-cloud-f3-fpga-as-a-service-instances_594057, 2018.

[3] Amazon, "Amazon ec2 f1 instances," https://aws.amazon.com/ec2/instance-types/f1/, 2019.

[4] ——, "Amazon ec2 pricing," https://aws.amazon.com/ec2/pricing/on-demand/, 2019.

[5] Xilinx, "Ultrascale+ fpgas product tables and product selection guide," https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf, 2018.

[6] ——, "Reaching new heights with the world's largest fpga," https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus-vu19p.html, 2019.

[7] P. Mell, T. Grance *et al.*, "The nist definition of cloud computing," 2011.

[8] N. Tarafdar, T. Lin, D. Ly-Ma, D. Rozhko, A. Leon-Garcia, and P. Chow, "Building the infrastructure for deploying fpgas in the cloud," in *Hardware Accelerators in Data Centers*. Springer, 2019, pp. 9–33.

[9] G. Dai, Y. Shan, F. Chen, Y. Wang, K. Wang, and H. Yang, "Online scheduling for fpga computation in the cloud," in *2014 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2014, pp. 330–333.

[10] K. Zhang, Y. Chang, M. Chen, Y. Bao, and Z. Xu, "Computer organization and design course with fpga cloud," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. ACM, 2019, pp. 927–933.

[11] A. A. Al-Aghbari and M. E. Elrabaa, "Cloud-based fpga custom computing machines for streaming applications," *IEEE Access*, vol. 7, pp. 38 009–38 019, 2019.

[12] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang, "Enabling fpgas in the cloud," in *Proceedings of the 11th ACM Conference on Computing Frontiers*. ACM, 2014, p. 3.

[13] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow, "Fpgas in the cloud: Booting virtualized hardware accelerators with openstack," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2014, pp. 109–116.

[14] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf, "Enabling fpgas in hyperscale data centers," in *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*. IEEE, 2015, pp. 1078–1086.

[15] J. M. Mbongue, F. Hategekimana, D. T. Kwadjo, and C. Bobda, "Fpga virtualization in cloud-based infrastructures over virtio," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 2018, pp. 242–245.

[16] J. Mbongue, F. Hategekimana, D. T. Kwadjo, D. Andrews, and C. Bobda, "Fpgavirt: A novel virtualization framework for fpgas in the cloud," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018, pp. 862–865.

[17] A. Vaishnav, K. D. Pham, D. Koch, and J. Garside, "Resource elastic virtualization for fpgas using opencl," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 111–1117.

[18] F. Hategekimana, J. M. Mbongue, M. J. H. Pantho, and C. Bobda, "Secure hardware kernels execution in cpu+ fpga heterogeneous cloud," in *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018, pp. 182–189.

[19] J. Mandebi Mbongue, D. Tchuinkou Kwadjo, and C. Bobda, "Flexitask: A flexible fpga overlay for efficient multitasking," in *Proceedings of the 2018 on Great Lakes Symposium on VLSI*. ACM, 2018, pp. 483–486.

[20] G. Schelle and D. Grunwald, "Exploring fpga network on chip implementations across various application and network loads," in *2008 International Conference on Field Programmable Logic and Applications*. IEEE, 2008, pp. 41–46.

[21] M. K. Papamichael and J. C. Hoe, "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*, 2012, pp. 37–46.

[22] N. Kapre and J. Gray, "Hoplite: Building austere overlay nocs for fpgas," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2015, pp. 1–8.

[23] P. Maidee, A. Kaviani, and K. Zeng, "Linkblaze: Efficient global data movement for fpgas," in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2017, pp. 1–8.

[24] Xilinx, "Ultrascale architecture and product data sheet: Overview," https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf, 2019.

[25] OpenCores, https://opencores.org/projects, 2020.

[26] Xilinx, "All programmable 7 series product selection guide," https://www.xilinx.com/support/documentation/selection-guides/7-series-product-selection-guide.pdf, 2018.

[27] N. Eskandari, N. Tarafdar, D. Ly-Ma, and P. Chow, "A modular heterogeneous stack for deploying fpgas and cpus in the data center," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2019, pp. 262–271.

[28] M. Asiatici, N. George, K. Vipin, S. A. Fahmy, and P. Ienne, "Virtualized execution runtime for fpga accelerators in the cloud," *Ieee Access*, vol. 5, pp. 1900–1910, 2017.

[29] S. A. Fahmy, K. Vipin, and S. Shreejith, "Virtualized fpga accelerators for efficient cloud computing," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2015, pp. 430–435.