IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 4, 2019, accepted March 16, 2019, date of publication March 22, 2019, date of current version April 5, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2906910

Cloud-Based FPGA Custom Computing Machines for Streaming Applications

AMRAN A. AL-AGHBARI AND MUHAMMAD E. S. ELRABAA
Computer Engineering Department, King Fahd University of Petroleum and Minerals (KFUPM), Dhahran 31261, Saudi Arabia
Corresponding author: Muhammad E. S. Elrabaa (erabaa@kfupm.edu.sa)

This work was supported by King Fahd University of Petroleum and Minerals.

ABSTRACT A novel platform for launching and using field-programmable gate arrays (FPFA) custom computing machines (CCMs) in clouds and data centers is proposed. Based on a developed FPGA virtualization scheme, it allows users to create independent computing services on network-attached standalone FPGAs. The interface of the virtual FPGA (vFPGA)-based CCM is automatically generated by a virtualization layer and based on the user's specifications. An FPGA hypervisor has been developed that can be easily integrated with any cloud management tool. It allows the users to launch/use/tear down vFPGA-based CCMs in a similar manner to conventional virtual machines (VMs). A complete prototype of the proposed platform has been realized and tested with a streamed image processing application. Its performance was 3–4x and ∼1.4–2.4x times better than an SW implementation on a VM and a powerful server, respectively. Compared with other platforms for FPGA attachment to a cloud or datacenter, the proposed platform has relatively low overhead in terms of FPGA resources while providing the highest level of abstraction and virtualization.

INDEX TERMS Reconfigurable computing, FPGAs, custom computing machines, cloud computing, streamed applications.

I. INTRODUCTION

On-cloud data processing is an important aspect of to-day's computations that serve other ecosystems such as IoTs and smart grid. Clouds offer efficient storage, sharing, and big data processing for these ecosystems. At the beginning of the cloud computing era, CPUs were used for all computing purposes. With the increasing demand on performance, accelerators such as GPUs and recently FPGAs have been extensively utilized in computations. CPU-based servers off-load compute-intensive tasks to GPUs and FPGAs to improve performance. Heterogeneous computing with FPGAs offers lower power consumption per operation compared to CPUs and GPUs. In comparison to traditional CPUs, the power consumption of FPGAs is 90% lower [1]. Additionally, FPGAs are well suited for packet processing systems. They have been reported to improve the bandwidth between virtual ma-chines (VMs) in a public cloud from 4Gbps to 25Gbps, with five to ten times less latency [2]. Excellent performance for AI, image processing, data compression as well as many other applications have been reported for FPGAs. Such improvements exceeded 300x for pattern matching, 200x for compression, and 100x for machine learning [3].

The associate editor coordinating the review of this manuscript and approving it for publication was Gian Domenico Licciardo.

Though FPGA vendors have developed compilers and libraries that allow creating and executing kernels on FPGAs similar to GPU kernels, FPGA virtualization remains necessary for FPGA usage in clouds. There is a need to integrate FPGAs into clouds and allow them to be managed in a similar way to other cloud resources (CPUs, storage and networking). Users should be able to instantiate them as independent computing machines and deploy applications on these FPGAs in a similar manner to virtualized CPUs. They should not have to instantiate a physical machine with an attached FPGA to accelerate their applications as it is the case with most current FPGA offerings in public clouds.

In this work, we introduce a cloud computing frame-work to provision virtual FPGAs as standalone custom computing machines (CCMs). A CCM is a network-attached virtual FPGA programmed with specific hard-ware application, receives data in its standard format without special formatting, performs computations, and sends back the results just like standard software functions on network servers. For instance, a CCM for image edge detection application accepts JPEG images and produces another JPEG image which the detected edges. The user sends and receives images in their original format through a direct connection with the CCM. CCMs are self-contained and self-controlled and do not require external servers or controllers.

The proposed FPGA-based CCM frame work for public clouds has the following novelties:

- It hides hardware details from the user. The developed CCMs receive data in its original raw format. A specially developed on-FPGA controller reformats the da-ta according to user specifications and follows the hardware I/O protocol.

- Unlike other works, computation in CCM is not man-aged or controlled by another CPU-based server. It is a standalone computing machine that starts computations once it receives new data. This makes it well suited for real-time computation and IoT computations.

- We introduce a cloud platform that manages CCMs and provision them as cloud computing services. Our cloud platform can be smoothly merged with existing cloud systems. We discuss the FPGA hypervisor and the software library used to manage and use CCM re-sources. The FPGA hypervisor is independent like any hypervisor in the cloud.

Next, related work on FPGA virtualization and attachment to clouds and datacenters is reviewed. Overview of the proposed framework is introduced in section 3 fol-lowed by a detailed explanation of the implementation in section 4. Experimental results and comparisons to other platforms are presented in section 5 followed by conclusions in section 6.

II. RELATED WORK

Most cloud-based FPGA virtualization efforts share the concept of dividing the physical FPGA into a fixed or static logic part (called the *shell*) and a configurable part (the *role*). The *shell* (1% up to 25% of the physical FPGA) abstracts the interface and performs management and the role constitute the virtual FPGA and holds the user's application. The *shell* contains the common components required by all applications such as communication controllers (PCI end point, network controller, off-chip memory controller), clock management, routing (if multiple vFPGAs are allowed) and reconfiguration management (if dynamic reconfiguration is supported). Researches refer to the static region using different names such as static logic [6], [8], RC2F [5], service layer [9], vendor logic [13], [14], Network Service Layer [28], FPGA hypervisor [7], [12] and *shell* [10], [11]. The role region is usually a dynamically reconfigurable region [8], [9], [13], [5]. Some researchers proposed a static *role* that would be combined with the static logic and synthesized to get the full FPGA bit stream [10], [22]. Compilers for this approach, compile an application (written in High-level language) into a SW part and FPGA part that executes the user-specified compute-intensive functions (kernels). The compiler generates the SW drivers and HW interfaces that enable the two parts to communicate using the node's PCIe bus. Several role regions can be merged to form one large region [23]. Migrating a design from one role to another was discussed in [21] though the measured migration time was around 1 S.

Physical FPGAs can be physically attached to a datacenter (DC) node's CPU via the PCIe, as a stand-alone networked resource, or both [2], [5], [6], [9], [10], [25]. Tight coupling between FPGAs and CPUs in compute nodes leads to several limitations:

1) The number of FPGAs in a DC is limited to the number of CPU nodes and PCIe slots per node,

2) FPGAs cannot be used independently from the CPU node they are attached to; i.e. CPUs must explicitly send/receive data and instructions to the FPGAs wasting both CPU's and FPGA's cycles,

3) In a cloud setting, customers actually instantiate two compute instances (one on a CPU and another on an FPGA),

4) Aggregation of several FPGAs to implement a large application becomes difficult and inefficient, as the data traffic between these FPGAs must go through the nodes (i.e. no direct communication between FPGAs). This is why some platforms utilize both interfaces [11] with a secondary network between FPGAs to enable running large applications on multiple FPGAs even though they are PCIe-attached. In [14] FPGAs are used as standalone network-attached computing resources, thus decoupling the number of FPGAs from the number of CPU nodes and PCIe slots per node.

Thus far, little has been done to abstract the I/O interfaces of cloud-attached FPGAs. Applications on virtual FPGAs have different data bus widths and several I/Os for control. The proposed *shells* in the literature are static and provide fixed *shell/role* interfaces to be used by different applications. AXI interface [29] is commonly used as a *shell/role* interface as it provides efficient handshaking and allows high data exchange rates. Amazon EC2 F1 instance [10] provides several AXI interfaces for the application. FIFO-based interfaces are also used in [5], [7], [13], [28], [37]. The empty and full signals of the FIFO allow both sides to know when the other side is busy. Asynchronous FIFOs can be used when the role and *shell* clocks are different.

The nature of hardware applications however, is to use reset and other control signals in addition to data buses. Each application has its own I/O specifications. This reveals the need for a controller that manages the traffic with the application and the need for abstracting the *shell/role* interface itself. In several virtualization plat-forms, the *shell/role* interface act as a DMA controller [13], [37]. This method is also used in GPU-like virtualization platforms that defines hardware kernels and restricts their interfaces to the off-chip memory [26]. In [33], [34], adding on-board processor or soft processor to the *shell* is proposed for orchestrating the hardware accelerator's execution. A manager executing on the processor man-ages the FPGA resources and the communication with the host over the PCIe. Still, all these methods require the application developer to redesign the hardware and adapt its interface to match the *shell/role* interface. In our previous work, we have completely abstracted the virtual FPGA I/Os [27] by integrating an auto-generated wrapper as an abstraction layer between the *shell* and the *role*.

FPGA virtualization (i.e. I/O abstraction) is the first step towards integrating them into public clouds. The second step is to integrate them into the cloud management framework. For PCIe-attached FPGAs, the host compute node would host

a virtual machine (VM) with the PCI-attached FPGA. The VM hypervisor is slightly modified to launch VM requests within machines that have PCI-attached FPGA [12]. Wang et. al. [30] demonstrated a different method in which, PCI-attached FPGA can be shared among several servers. The Xen virtual machine monitor (VMM) was used to provide FPGA access to all servers on the network and manage the PCIe traffic be-tween servers and hardware on the FPGA.

Several researchers proposed modifications to the popular OpenStack [4] cloud management system to enable integrating FPGAs as computing resources [8], [9], [12], [13]. Amazon has introduced Amazon FPGA Image (AFI) management tools [24], [25]. It comprised of eight command line tools for listing available FPGA slots, getting an instant image status, loading an image, clearing an image, starting virtual JTAG to debug the design, and get/set LEDs and switches of the board. Knodel [5] introduced the RC3E FPGA hypervisor which provides functions for device control, vFPGA control, data flow control that interacts with the off-chip (board) memory, bitstream loading, getting the status and setting configurations. Many other works defined similar functions [7], [33], [34], [37].

Other researchers proposed a different approach for HW-based acceleration in the cloud; the ASIC Cloud [15]. Several instances of the same accelerator are designed and fabricated as ASICs (Application-Specific Integrated Circuits) with all the necessary interfacing on the ASIC chip. Different accelerator ASICs are integrated on each board, and the boards are integrated into the CPU racks. Though these ASIC accelerators provide higher performance than their FPGA-based counterparts, they cannot be modified/updated. This is a waste of resources since all the accelerators that are not needed are not utilized.

In summary, current methods for FPGA attachment to the cloud attach the FPGA to a specific CPU via the PCIe bus or as stand-alone networked resources, or a combination of the two. PCIe attachment imposes a specific use case similar to GPUs and couple the number and usage of FPGAs to the number/usage of their host CPUs. This could be wasteful in a cloud setting. Existing compilers can compile an application into two parts; main kernel on the CPU (or a VM on the CPU) and compute intensive kernels on the FPGA. The compiler generates the necessary SW drivers on the CPU side and the HW interfaces on the FPGA side. The latter is usually a DDRx-based interface (i.e. assuming the HW kernel reads/write data to the off-chip DDRx memory) similar to a GPU interface. Stand-alone FPGA attachment decouples the number/usage of FPGAs from the CPU nodes, however, most of the existing solutions still impose a fixed inter-face to the application on the FPGA (mostly queues to/from off-chip DDRx memory). Hence, unlike the PCIe-attached FPGAs, the HW designer now must de-sign/implement the interface on his/her circuit's side. In addition, most existing solutions (for both types of attachments) do not provide intrinsic security measures, something that is essential in a public cloud environment.

In this work, we introduce an API-based (Application Programming Interface) framework for FPGAs attached as stand-alone resources in a public cloud setting. The proposed framework builds on and utilizes our FPGA virtualization scheme [27], and can be easily integrated with any cloud management system. Additional circuitry has been added to the vFPGA platform to secure all traffic between the HW application and the user. Users can design their own circuit blocks or select ready-made circuit IPs (intellectual properties), assemble their HW custom application from these circuit blocks, and then use the developed to automatically generate the interface (with built-in security) to the vFPGA's static logic (i.e. *shell*). The developed APIs are then used to manage, launch, use, and release vFPGA-based custom computing machines (CCMs) on the cloud.

III. OVERVIEW OF THE PROPOSED PLATFORM

In our proposed framework, Fig. 1, the cloud offers virtual FPGAs (vFPGAs) as CCMs that can be accessed within the cloud via a regular socket interface. The CCM is highly abstracted such that it can take streamed data directly without any pre-processing. Input/Output data formatting/reformatting and interfacing is carried out by the on-FPGA *shell*. The framework includes the following components:

1) **The cloud infrastructure**: which consists of FPGA hypervisor, image creator and other components. The FPGA hypervisor is used to manage vFPGAs resources, CCM images (bitstreams) launching and termination. It acquires CCM images from the cloud storage. The re-sources database stores information about CCMs and vFPGA for the management process.
2) **FPGAs**: which are connected to the internal cloud network. Each FPGA contains a static logic *shell* and one or more virtual FPGAs (vFPGAs) that act as compute nodes on the cloud when configured with CCM images. The static logic on an FPGA represents the hypervisor's back-end for the vFPGAs on that FPGA.
3) **Software library**: that defines the necessary APIs to manage and use vFPGAs. Two types of APIs are provided; socket APIs and message passing APIs. APIs such as launch and release CCMs are provided by the FPGA hypervisor, while other APIs such as configuring vFPGAs, reading status registers, setting the client's IP address, etc. are provided by the FPGA hypervisor's back-end (i.e. the on-FPGA *shell*).
4) **The image creator**: which receives the user's request, creates the new CCM image, updates the resources' database and stores the image in the cloud storage. The platform introduces CCM as a service in which CCMs can be implemented and sold by the cloud operator (CO) or a 3rd party to the client.
5) **CCM Image management** is part of the cloud storage management which stores and read CCM images to/from cloud storage.

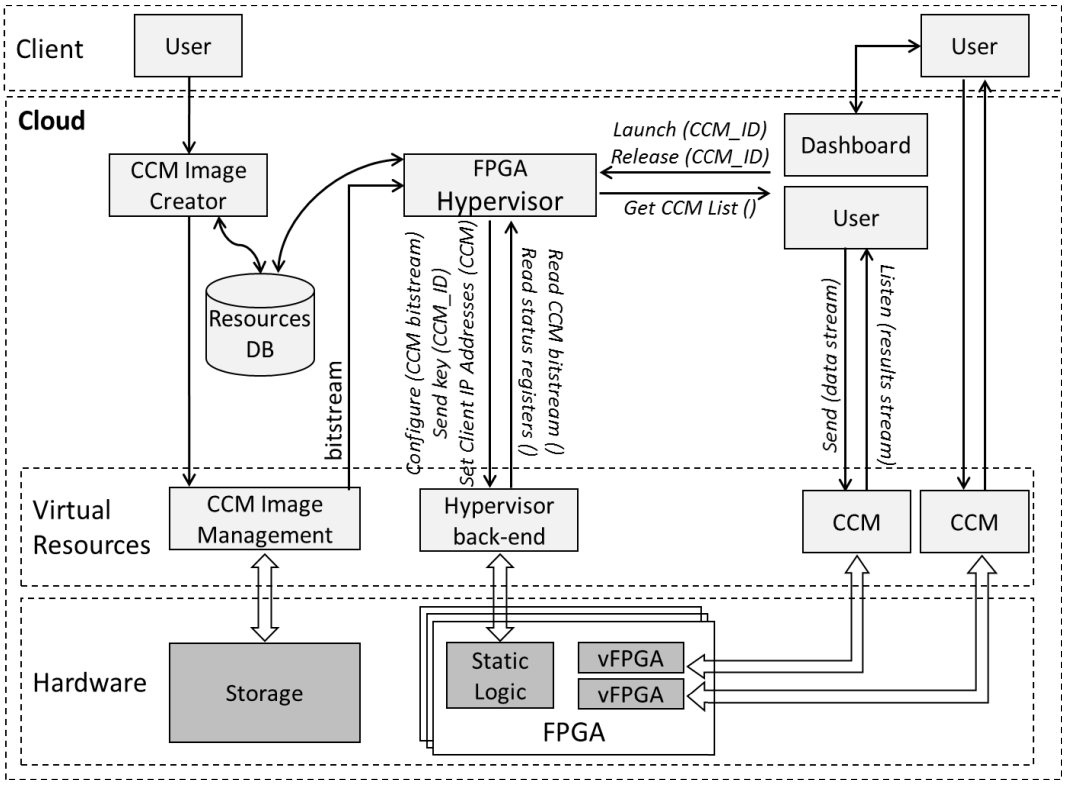

FIGURE 1. Proposed framework for on-cloud FPGA-based custom computing machines.

6) **The user** could be internal or external user. Internal users are applications running on the cloud or other CCMs. External users are applications running outside the cloud. They can interact with the on-cloud CCMs using the cloud's dashboard's provided IP address. The cloud management SW (its gateway) then translates this IP address to the CCM internal IP address.

IV. DETAILED IMPLEMENTATION

A. FPGA VIRTUALIZATION

The FPGA virtualization platform for deploying FPGA CCM-as-a-service on clouds, is shown in Fig.2. The FPGA is divided into static logic and several dynamically reconfigurable regions called virtual FPGAs (vFPGAs). The *static logic (shell)* contains the following permanent (i.e. static) hardware required to serve vFPGAs:

1) *A network controller* (**NC**): that implements all the low-level networking protocols (i.e. DHCP, ARP and ICMP protocols) for all vFPGAs and the shell. It responds to ping and ARP requests to announce existence and implements the DHCP protocol to obtain dynamic IP addresses. The network controller establishes TCP sessions with vFPGAs or the on-FPGA reconfiguration manager (**RM**). It also acts as a local switch; extracts payloads of incoming packets and forward them to the targeted vFPGA or **RM**, and receives computation results from vFPGAs, packetize them and send them the TCP session's destination IP address.

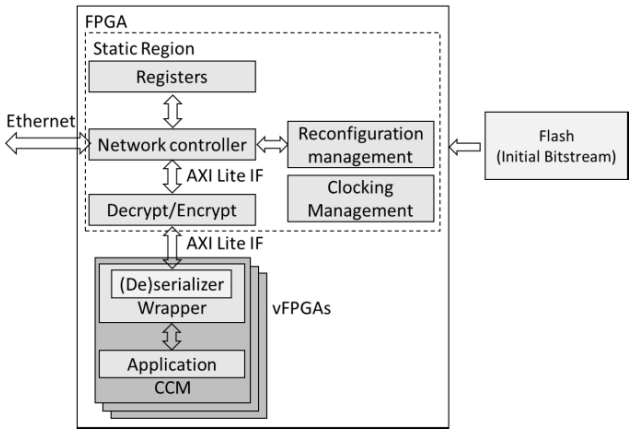

FIGURE 2. FPGA virtualization platform. FPGA is divided into static logic (shell) and several virtual FPGAs. Each vFPGA has IP/MAC addresses and TCP sessions are used to access them.

2) *Registers*: such as user IP addresses and Encryption keys. Only one user at a time can connect to each CCM. IP addresses and keys of those users are configured by the FPGA hypervisor.
3) *Off-chip flash*: is used to boot the whole FPGA. It stores the initial bitstream of the whole physical FPGA which contains the static logic with empty vFPGA regions.
4) *A reconfiguration manager* (**RM**): that to safely configures vFPGAs with their respective CCM images.

5) *A clock management* (**CM**) unit: that supplies all the required clock signals to the static region, the network controller and the vFPGAs.
6) *A Decrypt/Encrypt* engine: that is placed between the network controller and the CCMs to enable secure computations. It manages all the user-provided keys that are used by each CCM for each session, decrypts the received users' data before forwarding them to the CCMs and encrypts the CCM results before forwarding them to the network controller. The interfaces between the network controller, the Decrypt/Encrypt engine and the CCMs are all AXI lite interfaces.
7) *CCM wrapper*: That is generated by a special virtualization layer (script) based on the user's specifications. The wrapper contains a serializer which pre-format the data and apply it according to the user-specified interfacing protocol. It also contains a deserializer that re-format the CCM's output data. The wrapper abstracts the CCM as a compute machine that have its own MAC and IP addresses, receives data, and produces results over socket interface. The data itself does not contain timing or clocking information, the wrapper provides all that.

B. FPGA HYPERVISOR

The FPGA hypervisor manages vFPGA resources and CCM images. It implements APIs for launching, using and releasing vFPGAs and keeps track of available vFPGAs and CCM images by updating the *resources DB*. The *resources DB* is a database that stores vFPGA and CCM images management information such as occupied/free vFPGA resources, user-CCM assignment, vFPGA-CCM assignment, etc. The static logic in each FPGA manages vFPGAs and represents the hypervisor's back-end. Table 1 lists the main functions provided by the software library in the FPGA hypervisor which are described below.

TABLE 1. Main functions (provided as APIs) of the FPGA hypervisor.

Function name	Communication	Category
Send (data stream) ***Listen (results stream)***	Socket (TCP Stream)	CCM functions
Launch (CCM_ID) ***Get CCM List ()*** ***Release (CCM_ID)***	Message Passing	FPGA hypervisor functions
Configure (CCM bitstream) ***Read CCM bitstream ()***	Socket (TCP Stream)	Hypervisor back-end functions
Read status registers () ***Send key (CCM_ID)*** ***Set client info (Sender IP Address, Receiver IP Address)*** ***Set Parameters()***	Socket (UDP)	Hypervisor back-end functions

```python
import socket
BUFFER_SIZE = 1024

def Send_data(SERVER_IP, SENDING_PORT, data):
    conn_server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    conn_server.connect((SERVER_IP, SENDING_PORT))
    conn_server.send(data)
    conn_server.shutdown(socket.SHUT_RDWR)
    conn_server.close()

def Listen_to_results(SERVER_IP, RECEIVING_PORT):
    conn_server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    conn_server.connect((SERVER_IP, RECEIVING_PORT))
    data=''
    while len(data)<100
        data = data + conn_server.recv(BUFFER_SIZE)
    conn_server.shutdown(socket.SHUT_RDWR)
    conn_server.close()
    return data
```

FIGURE 3. Python implementations for the functions *"Send (data stream)"* and *"Listen_to_results (data stream)"*. Both functions use TCP stream socket and require the CCM IP address and port number.

1) **CCM functions**: Accessing a CCM is done through only two functions for transmitting and receiving data. An implementation example of two functions using python is shown in Fig.3. The function "Send (data stream)" establishes a TCP stream session and send the data stream over the session. The function "Listen (results stream)" establishes a listening TCP stream session and collect the results. The user should call the listener first then sends his/her data. The CCM hardware receives a reset signal with the creation of each TCP session.
2) **FPGA hypervisor functions**: Users use the hypervisor through message-passing APIs. The function," Launch *(CCM_ID)"* sends a message to the hypervisor to launch a CCM. The hypervisor would then obtain the CCM_ID from the database and assign a suitable free vFPGA. Then, it fetches the appropriate bitstream image from the storage and uses the function *"Configure (CCM bitstream)"* to download the CCM image. Finally, the hypervisor sends a message to the user with the IP address of the launched CCM. The function," *Release (CCM_ID)"* sends a message to the hypervisor to release the vFPGA resources of a CCM. When the function" Get *CCM List ()"* is called, the hypervisor using the resources DB, builds a CCM list their unique CCM_IDs and description.
3) **Hypervisor back-end functions**: These are socket functions between the hypervisor front-end and back-end. The function, *"Configure (CCM bitstream)"* downloads a partial bitstream that represents a CCM image on the FPGA. The function *"Read CCM bitstream ()"* reads back the CCM bitstream which is useful for supporting CCM migration. The function *"Read status registers ()"* reads information about the running CCM status. The function *"Send key (CCM_ID)"* changes the encryption/decryption key of the CCM. The function *"Set client info (Sender IP Address, Receiver IP Address)"* changes the sender and the

receiver IP addresses of the CCM. The function *"Set Parameters ()"* is used to configure some registers with specific values. One example is the frequency register that determine the CCM operating frequency.

1) A TYPICAL CCM USE-CASE SCENARIO

The following is a typical scenario for launching, using, and terminating a vFPGA-based CCM. It illustrates how the platform's APIs work. The user, who wants to launch and use a specific CCM, issues the four commands listed in Fig.4.

```
User commands:
  IP Address = Launch CCM (CCM_ID);
  results = Listen ((IP Address , Receiving PORT NO));
  Send ((IP Address, Sending PORT NO), data stream);
  Terminate CCM (CCM_ID);
```

FIGURE 4. Scenario of using a vFPGA-based CCM in a public cloud using the proposed platform. The user issues four commands to launch, send data, receives results, and terminate CCM platform.

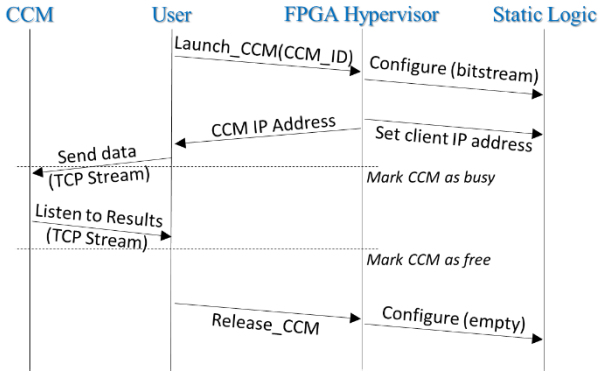

FIGURE 5. Timing diagram showing the message sequence for using a vFPGA-based CCM in the cloud using the proposed platform.

The corresponding actions performed by the platform are shown in the timing diagram of Fig.5 and are explained below (please refer to Fig.1 for the platform's components):

1. **IP Address = Launch CCM (CCM_ID)**
   - The CCM information and available free vFPGAs are obtained from the Resources DB, a suitable vFPGA is selected, and the CCM image is obtained from "CCM image management".
   - The hypervisor gets the IP address and port number of the specific FPGA, and other network parameters from the Dynamic Host Configuration Protocol (DHCP) server and then executes the internal function "***Send ((IP Address, Port_no), bitstream)***".
   - The hypervisor back-end configures the vFPGA with the CCM image
   - The hypervisor internally issues the function "***Set Client IP Addresses (Sender IP Address, Receiver IP Address)***" to configure the sender and receiver client IP addresses in the hypervisor back-end.
   - The hypervisor back-end opens a listener to receive CCM inputs and starts another TCP session for sending the results.
   - The hypervisor returns the CCM IP address to the user. The hypervisor-back end is never revealed to the user.
2. **results = Listen ((IP Address, Receiving PORT NO))**
   - The function is executed in the user machine to start the listening session. It is usually executed as a new thread so the program can overlaps sending data and receiving results.
3. **Send ((IP Address, Sending PORT NO), data stream)**
   - The user sends the data to the CCM. The function is executed in the user machine. It establishes a TCP stream session, sends the data to the CCM and terminates the session.
4. **Terminate CCM (CCM_ID)**
   - The hypervisor executes internal function "**Send ((IP Address, Port_no), empty bitstream)**"
   - The hypervisor back-end configures the vFPGA with the blank CCM image
   - The hypervisor back-end clear the registers of the sending and the receiving IP addresses.
   - The hypervisor updates the "Resources DB" and marks the vFPGA resource free.

2) CCM SHARING

A CCM can be shared among several users by interleaving computation sessions. The computation session is an atomic operation that cannot be interrupted. When a user uses a CCM, the hypervisor prevents other users from using it. When the current user's TCP session(s) to the CCM terminate, another user can request the same CCM and the hypervisor restricts its use to the new user for one session and so on. With each session the whole CCM is reset. The CCM's serializer and deserializer take care of flushing all results out before terminating the session.

3) USER DATA SECURITY

If the user requires a secure channel to the CCM, (s)he exchanges a symmetrical encryption key with the FPGA hypervisor using Diffie–Hellman key exchange. Then, the hypervisor uses the function "Send key (CCM_ID)" to send the key to the hypervisor's back-end. The encryption and decryption engine uses that key to decrypt incoming data and encrypt outgoing results. The hypervisor front- and back-ends take care of removing the key with each change in the sender's and receiver's IP.

4) CCM CLUSTERS ON MULTI-vFPGA

A cluster of network connected CCMs can be created and saved as a new CCM. CCM network can be built by carefully setting the sending and receiving IP addresses of each CCM

in the cluster. For example, an FPGA chain can be created by setting the receiving IP address of each FPGA in the chain as a sending IP address for its previous vFPGA. The receiving address of the first vFPGA and the sending addresses of the last vFPGA in the chain becomes addresses for the resulted CCM. The new CCM information is stored in the Resources DB with pointers to the information of other CCMs constructing it.

C. THE IMAGE CREATOR

The platform introduces CCM as a service in which CCMs can be implemented and sold by the cloud operator (CO) or other third parties to the cloud's tenants. The image creator, Fig.6, contains the required software tools to create CCM images. A CCM image is a partial bitstream FPGA configuration file. The user sends a hardware design using a hardware description language (HDL) such as Verilog or VHDL along with two text-based files; an XML file for the design's IOs, and a file that describes how data is going to be applied to the hardware. The latter is specified using the standard HW verification language Vera [32]. The Image creator uses these two input files to generate an additional HDL file containing a wrapper for the design using the methodology in [27]. The wrapper's interface matches the shell/role interface of the virtualization platform. The wrapper also contains the serializer that applies input data to the hardware according to the Vera description provided by the user. The wrapper is synthesized with the user's hardware instantiated within it, and several partial bitstreams are generated to match several types of vFPGA. The Image creator updates the Resources DB by adding the new CCM data which includes vFPGA type and file names and then it sends CCM image files to the cloud storage. Fig.6. illustrates how the Image creator works.

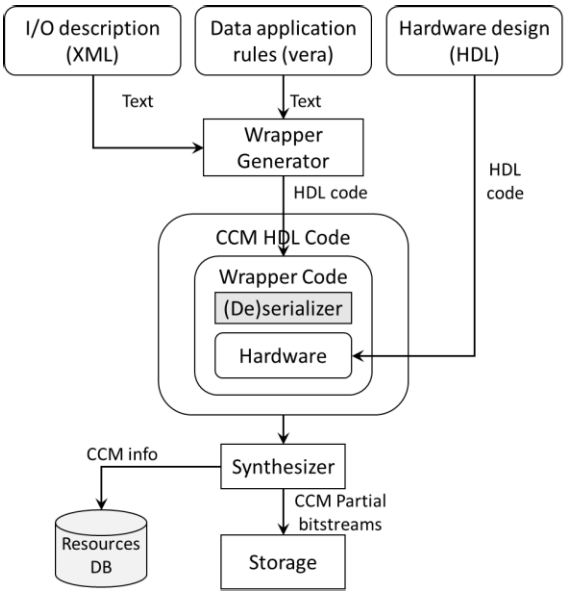

FIGURE 6. Illustration of the CCM image creator's components and how they generate the design's partial bitstream files.

V. EXPERIMENTAL RESULTS

A secure image edge detection (ED) CCM is used as test case to show how a CCM can be accessed as a cloud service using the proposed framework in a similar manner to accessing a software as a service. The CCM receives a JPEG image and produces another JPEG image with the detected edges as illustrated in Fig.7. To this end, we built a CCM for the application as well as the FPGA virtualization platform. We have also developed a pure SW implementation of the same application using standard Python libraries with TCP stream sockets for the SW application interface. Two versions of the software implementation were run; one on a server and another on a virtual machine. We also designed another software to act as a user that requests the application's service. The user uses TCP stream socket interface to request the service from CCM or the SW implementations. The experimental setup is shown in Fig. 8. The FPGA is a Xilinx Virtex 6 XC6vlx550t FPGA. The server machines have dual socket Intel 8-core Xeon CPUs running at 3.00GHz, 16GB of RAM, and 64bit-linux Ubuntu 16.04LTS. The VM machine is a Virtualbox virtual machine with 4 GB RAM, bridged Ethernet and 64bit-linux Ubuntu 16.04LTS. The user's workstation has a core-i7 CPU running at 3.00GHz, 16GB of RAM, and 64bit-windows 10.

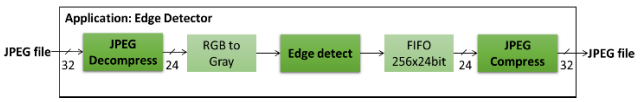

FIGURE 7. The implemented image edge detection application encryption/decryption on input/output data is performed by the static logic.

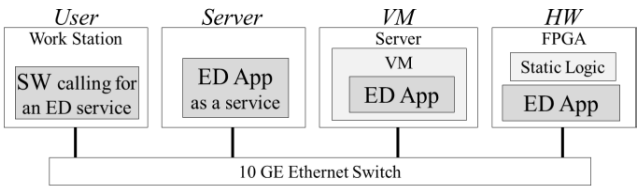

FIGURE 8. Timing the experimental setup with several versions of the secure edge detection (ED) application.

A. THE FPGA-CCM IMPLEMENTATION

The Static Logic (Fig. 2) utilizes the TinyAES [16] to build counter mode AES encryption/decryption blocks (AES-CTR). The test case hardware is designed by integrating several open-source cores [16] (JPEG decompressor, Canny edge detector and JPEG encoder) to make the edge detector CCM. The JPEG encoder has a fixed delay per pixel while the decompressor's delay varies according to the input image and dominates the total image processing time. FIFOs were used in between different blocks to overlap their operation as they had different bandwidths. The wrapper was generated, co-synthesized with the CCM, and both were placed on a

```
#Server side (software compute node)
from Crypto.Cipher import AES
import numpy
import cv2
mode = Crypto.Cipher.AES.MODE_CTR
ctr_encr=Crypto.Util.Counter.new(128,initial_value=long(variables.IV.encode("hex"),16)
ctr_decr=Crypto.Util.Counter.new(128,initial_value=long(variables.IV.encode("hex"),16))
AES_encr=Crypto.Cipher.AES.new(variables.key, mode, counter=ctr_encr)
AES_decr=Crypto.Cipher.AES.new(variables.key, mode, counter=ctr_decr)
…
def compute(data_in):
  img     = AES_decr.decrypt(data_in)
  nparr   = numpy.fromstring(img, numpy.uint8)
  img_np  = cv2.imdecode(nparr, cv2.IMREAD_COLOR)
  edges   = cv2.Canny(img_np,100,200)
  img_np2 = cv2.imencode(".jpg", edges)
  return AES_decr.encrypt(img_np2[1].tostring())
```

FIGURE 9. Snapshot of the software version of the secure image edge detection application written in python using standard SW libraries.

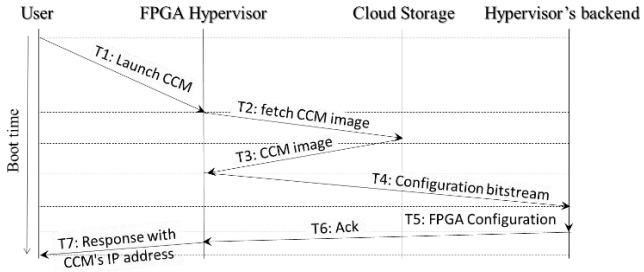

FIGURE 10. Different boot time components of an vFPGA-CCM.

vFPGA. The resource utilization report for the ED CCM with a 10GE network interface is shown in Table 2.

TABLE 2. Resource utilization of the cloud-based FPGA ED CCM.

	LUTs	FFs	RAMs	DSPs
Static logic	12,462	10,990	161	0
Total CCM	45,661	41,659	261	560
- decode	11,457	8,428	9	21
- detect	3,262	3,833	18	0
- encode	26,761	29,966	30	560
Total	58,123	52,649	422	560

B. THE SOFTWARE IMPLEMENTATION

The software implementation of the secured image edge detection is written in Python and launched on two separate platforms, a server and a virtual machine. To ensure the best throughput, the application was built using standard libraries to build the application; the standard Python Cryptography Toolkit (PyCrypto) [17] for encryption/decryption, and the computer vision (OpenCV) library [18] for the Canny edge detection. A snapshot of the code is shown in Fig.9. The service first decrypt the received JPEG image, stores it in an array to pass it to the image decoder. Then, Canny edge detect function from OpenCV library detects the edges and the resulted image is encoded again to produce a JPEG image. Finally, the resulted JPEG image is encrypted to be sent to the user.

TABLE 3. Image processing times and throughputs of the three implementations of the secure image edge detection application.

Input Image Size (Bytes)	Average Image Processing Time (ms)			Average Throughput (Mbits/Second)		
	vFPGA-CCM	VM	Server	vFPGA-CCM	VM	Server
58,962	4.30	17.38	9.67	109.63	27.14	48.76
72,618	4.78	20.58	10.64	121.43	28.22	54.62
84,644	5.07	22.16	12.00	133.52	30.56	56.43
114,726	7.42	25.95	14.83	123.69	35.37	61.90
128,573	8.50	35.09	15.26	121.01	29.31	67.42
163,301	10.05	33.46	18.71	129.98	39.05	69.81
195,211	11.58	35.27	19.97	134.81	44.28	78.21
201,071	12.28	36.24	19.93	131.01	44.39	80.71
266,529	14.06	45.23	23.52	151.67	47.14	90.65
864,475	50.00	206.76	99.14	138.33	33.45	69.76

TABLE 4. Boot time delay components for vFPGA-CCMs with various image (bitstream) sizes. Internal configuration access port's speed is ~400MB/s.

Bitstream size (MB)	Timing Components							
	T1 (ms)	T2 (ms)	T3 (ms)	T4 (ms)	T5 (ms)	T6 (ms)	T7 (ms)	Total Boot time (ms)
1	250	5	9	9	3	5	250	530
5			44	44	13			611
10			89	89	25			713
15			133	133	38			814
20			178	178	50			916

TABLE 5. Boot time (in seconds) for various virtual machine configurations implemented on Openstack.

VM Size	Virtual cores	RAM	Disk	Ephemeral Storage	Boot time(s)
X Large	8	16 GB	10 GB	160 GB	52
Large	4	8 GB	10 GB	80 GB	46
Medium	2	4 GB	10 GB	40 GB	41
Small	1	2 GB	10 GB	20 GB	34
Tiny	1	512 MB	1 GB	0 GB	30

C. PERFORMANCE COMPARISONS

For all implementations of the test case application, the client sends encrypted images and receives encrypted edge-detected images over TCP stream sessions using two parallel threads (one for sending images and another for receiving the resulting images). Python Stream Socket library was used for that purpose as it provides throughputs close to the theoretical line bandwidth. Ten JPEG images (the 1st nine are 640x480 pixels while the 10th image is 1920x1080 pixels) were used in the evaluation. The variation in size reflects the compression ratio which varies according to the image contents. Each

TABLE 6. Comparison with notable platforms for FPGA-based processing in clouds or datacenters.

	Configuration/Attachment	Clustering	Interface Abstraction	DDRx	Throughput (MB/s)	Total Area
[5] RC3E	PCIe-attached to a host		Low		798	7,082 LUTs +6,974 FFs
[37] DyRACT	PCIe-attached to a host		Low	✓	864	16,157 LUTs + 19,453 FFs
[6] Fahmy	PCIe-attached to a host		Low	✓	6,800‡	30,324 LUTs+60,648 FFs
[33] Asiatici	PCIe-attached to a host		Low	✓	-	Not reported
[34] Asiatici	PCIe-attached to a host		Low	✓	864	16,965 LUTs + 20,426 FFs
[13] Hyperscale	Ethernet-attached to a host	✓	Medium	✓	-	44K LUTs
[14] IBM's Disaggregated	Standalone, Network attached		Low	✓	1,188 on 10GE	58,128 LUTs + 16,256 FFs
[28] IBM's Net-attached	Standalone, Network attached	✓	Medium		-	90,747 LUTs + 103,348 FFs
[12] Tarafdar	PCIe or Ethernet-attached to a host	✓	None	✓	118.5 on 1GE	62,344 LUTs + 7,124 FFs
[10] Amazon	PCIe-attached to a host		None	✓	-	Not reported
[11] MS Catapult	PCIe-attached to a host + Network attachment + FPGA Network	✓	None	✓	-	39,560 ALMs (≈ 80,000 LUTs + FFs)
[8] Byma	Standalone, Network attached		None	✓	100 on 10GE	28,711 LUTs + 29,327 FFs
[9] Chen	PCIe-attached to a host		Low	✓	1,280	13,156 LUTs + 26,312 FFs
[35] MS FENIKS	PCIe or Ethernet-attached to a host		Low	✓	6,000‡	~22,438 ALMs (≈ 46,000LUTs + FFs)†
[36] RIFFA	PCIe-attached to a host		Medium		3,640‡	15,862 LUTs + 14,875 FFs (Xilinx), 15,182 ALUTs + 13,418 FFs (Altera) (without PCI logic)
[15] ASIC Clouds	PCIe-attached to a host + Standalone Network attachment		Medium	✓	-	Not reported
[38] JetStream	PCIe-attached to a host	✓	Medium		33.5	8,571 LUTs + 6,955 FFs
This work	Standalone Network attachment	✓	Full		113 on 1GE, 938 on 10GE	27,504 LUTs + 30,514 FFs

† Reported as 13% of Altera Stratix V resources.

‡ Communication interface speed with very wide data words (128-256-bit wide).

image is encrypted using AES128-CTR and sent to the edge-detection service under test over the socket interface. The edge-detection service decrypts the image, decodes it, does edge detection, encodes the detected-edge image, encrypts the resulted image and returns it to the sender. Each image was sent (i.e. streamed) 100 times for each service implementation and the total time for each image processing was measured as the time between receiving the 1st packet of the input image and the time the last packet of the processed image is sent. Then the 100 time measurements were averaged to get the average processing time for each image. The average throughput was measured in a similar manner. Table 3 shows the detailed performance of the three service implementations for the 8 test images. As expected, the physical server had ∼2x the VM's performance due to the virtualization overhead. The vFPGA-based CCM however, achieved 3∼4x and ∼1.4–2.4x speed up over the VM and the physical server implementations, respectively.

D. BOOT TIME COMPARISONS

A vFPGA-CCM boot time is measured from the time the user sends a "**Launch a CCM**" request to the time (s)he

receives a response with the launched CCM IP Address as illustrated in Fig. 10. The component of this delay are message passing delays (e.g. the launch request, the response with the IP address, etc.), fetch the CCM image from the cloud storage and sending it to the FPGA hypervisor's back-end, and the FPGA configuration time by the hypervisor's back-end. Large FPGAs have an average bitstream file size of ∼10 megabytes, resulting in ∼25 milliseconds average configuration time through the internal configuration access port (ICAP) [20]. Table 4 shows the different delay components and the total vFPGA-CCM boot times for several sizes of CCM image (i.e. bitstream files) sizes. Compared to VM booting times, Table 5, the vFPGA boot times are in hundreds of milliseconds while the VMs' are in tens of seconds (i.e. 100X). The VMs in Table 5 were booted with only a CirrOS image (a very lightweight 12-MB version of Linux) on OpenStack cloud. Booting requests were issued from a client on the same LAN. A major difference between conventional VMs and vFPGA-CCMs is that for the vFPGA-CCMs, the only difference in boot time between different CCMs' images is the configuration time. VMs' boot time is highly dependent on VM specifications as illustrated in Table 5.

E. COMPARISONS WITH OTHER PLATFORMS

Table 6 below shows a comparison of the proposed FPGA-based cloud CCM platform with other notable platforms with similar purpose. The configuration/attachment refers to how the FPGA is attached to the cloud (or datacenter) and how it is used. FPGA-based accelerators are attached to a host CPU via the PCIe bus and cannot be used independently from the host (i.e. in a cloud environment, the user has to instantiate two compute instances). Depending on their shells, FPGAs attached to the data center's network can be used on their own (i.e. standalone) or still need to work in tandem with a host CPU that runs the main application and calls FPGA acceleration functions over the Ethernet. The latter option also requires two compute instances. Network-attached standalone FPGAs act as servers (i.e. can be used by multiple users/applications). Microsoft's Catapult provides all types of attachments and configuration at a staggering logic cost [11].

The clustering column indicates whether several FPGAs can be connected directly to run large applications without having the data going through CPU nodes. JetStream is the only PCI-attached FPGA that allows vFPGA to vFPGA connection. The IF (Interface) abstraction column reflects the level of abstraction for the application interface. Our proposed platform can receive data in their original format so it provides full abstraction. Medium abstraction is provided by FIFO interfaces. A platform with a low abstraction is one that requires users to adapt their design to its fixed interface. Platforms that require the users to develop custom interfaces have no IF abstraction at all. The DDRx column specifies if a platform's shell has a DDRx interface. We have not opted for this option as it increases the static logic area significantly and it is not crucial for streamed applications. The throughput column lists the effective communication throughput of the different platforms' interfaces. For many works it just represent the raw communication throughput. Users may not be able to adapt their design to take advantage of the high commination throughput. E.g. a PCIe interface that delivers ∼ 4 GB/s will do so as 128-bit wide data at 250 MHz or 256-bit wide data at 125 MHz If the user's core cannot process this data, it will have to throttle back the communication link by applying back pressure. Our platform completely abstracts the interface and apply data to the user's design at the required width and frequency.

The ASIC cloud was included in the comparison because it provides ASIC custom computing machines for the cloud. IBM's network-attached FPGAs [28] is the closest work to our work. Its IF abstraction is medium because it introduces fixed FIFO-based interfaces and the data formatting and the computation control is completely left for the application's designer to design. The table shows that standalone CCMs with abstracted data interface are not introduced by other FPGA virtualization platforms. It also shows that our proposed platform provides ultimate flexibility with a relatively low overhead.

VI. CONCLUSIONS

A platform for launching and using CCMs on virtualized FPGAs in clouds and data centers has been developed. Physical FPGAs are attached to the cloud's network and can host several vFPGAs. A complete FPGA hypervisor with was developed. Using a very simple API interface, the front-end runs on the cloud and allows users to generate, maintain, and retrieve CCMs images (i.e. CCMs FPGA configuration files). The hypervisor's backend is implemented as static logic (i.e. HW) on the FPGAs and it is responsible for configuring the vFPGAs with CCM images and setting up the required routing tables on the FPGA to enable multiple traffic to/from vFPGAs co-hosted on the same FPGA. The developed FPGA virtualization layer automatically generates the physical HW interface to the user's circuitry residing on the vFPGA based on the user's specifications. A complete prototype of the proposed platform has been realized and tested with a streamed secure image processing application. Compared to conventional VM-based and physical server-based implementations, the performance was 3∼4x and ∼1.4–2.4x times better, respectively. Comparison with other platforms for FPGA attachment to a cloud or datacenter has been carried out. It shows that our platform has a relatively low overhead in terms of FPGA resources while providing highest level of abstraction and virtualization.

REFERENCES

[1] C. Kachris and D. Soudris, "A survey on reconfigurable accelerators for cloud computing," in *Proc. 26th Int. Conf. Field Program. Logic Appl. (FPL)*, Lausanne, Switzerland, Aug./Sep. 2016, pp. 1–10.

[2] M. Russinovich and M. Branscombe. (Jun. 8, 2018). *FPGAs and the New Era of Cloud-Based 'Hardware Microservices'*. [Online]. Available: https://thenewstack.io/developers-fpgas-cloud/

[3] IBM Research China. (Nov. 19, 2014). *SuperVessel Cloud*. [Online]. Available: http://research.ibm.com/labs/china/supervessel.html

[4] Opensource. *OpenStack Cloud Management Suite*. Accessed: 2015. [Online]. Available: https://www.openstack.org/
[5] O. Knodel and R. G. Spallek, "Computing framework for dynamic integration of reconfigurable resources in a cloud," in *Proc. Euromicro Conf. Digit. Syst. Design*, Funchal, Portugal, Aug. 2015, pp. 337–344.
[6] S. A. Fahmy, K. Vipin, and S. Shreejith, "Virtualized FPGA accelerators for efficient cloud computing," in *Proc. IEEE 7th Int. Conf. Cloud Comput. Technol. Sci. (CloudCom)*, Vancouver, BC, Canada, Nov. 2015, pp. 430–435.
[7] H. L. Kidane, E. B. Bourennane, and G. Ochoa-Ruiz, "NoC based virtualized accelerators for cloud computing," in *Proc. IEEE 10th Int. Symp. Embedded Multicore/Many-Core Syst.-Chip (MCSOC)*, Lyon, France, Sep. 2016, pp. 133–137.
[8] S. Byma, J. G. Steffan, H. Bannazadeh, A. Leon-Garcia, and P. Chow, "FPGAs in the cloud: Booting virtualized hardware accelerators with OpenStack," in *Proc. IEEE 22nd Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2014, pp. 109–116.
[9] F. Chen *et al.*, "Enabling FPGAs in the cloud," in *Proc. 11th ACM Conf. Comput. Frontiers (CF)*, 2014, pp. 1–10.
[10] Amazon. (2017). *Amazon EC2 F1 Instances*. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/
[11] A. Putnam *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," *IEEE Micro*, vol. 35, no. 3, pp. 10–22, May/Jun. 2015.
[12] N. Tarafdar, N. Eskandari, T. Lin, and P. Chow, "Designing for FPGAs in the cloud," *IEEE Design Test*, vol. 35, no. 1, pp. 23–29, Feb. 2018.
[13] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf, "Enabling FPGAs in hyperscale data centers," in *Proc. IEEE 12th Int. Conf. Ubiquitous Intell. Comput. IEEE 12th Int. Conf. Auton. Trusted Comput. IEEE 15th Int. Conf. Scalable Comput. Commun. Associated Workshops (UIC-ATC-ScalCom)*, Beijing, China, Aug. 2015, pp. 1078–1086.
[14] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf, "Disaggregated FPGAs: Network performance comparison against bare-metal servers, virtual machines and Linux containers," in *Proc. Int. Conf. Cloud Comput. Technol. Sci. (CloudCom)*, Dec. 2017, pp. 9–17.
[15] I. Magaki, M. Khazraee, L. V. Gutierrez, and M. B. Taylor, "ASIC clouds: Specializing the datacenter," in *Proc. 43rd Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 178–190.
[16] D. Lundgren. (2010). *JPEG Encoder Verilog*. [Online]. Available: https://opencores.org/
[17] *STREAM Socket Programming on Python*. Accessed: 2017. [Online]. Available: https://docs.python.org/2/howto/sockets.html
[18] *Open Source Computer Vision (OpenCV) for Python*. Accessed: 2017. [Online]. Available: https://docs.opencv.org
[19] M. Leonhard. (2017). *CloudPing.info for Amazon Web Services Available in Several Regions/Cloudping*. [Online]. Available: http://www.cloudping.info/
[20] Xilinx. (May 3, 2010). *Partial Reconfiguration User Guide*. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/ug702.pdf
[21] O. Knodel, P. R. Genssler, and R. G. Spallek, "Migration of long-running tasks between reconfigurable resources using virtualization," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 4, pp. 56–61, Jan. 2017. doi: 10.1145/3039902.3039913.
[22] Q. Zhao, M. Amagasaki, M. Iida, M. Kuga, and T. Sueyoshi, "Enabling FPGA-as-a-service in the cloud with hCODE platform," *IEICE Trans. Inf. Syst.*, vol. E101-D, no. 2, pp. 335–343, 2018.
[23] O. Knodel, P. R. Genssler, and R. G. Spallek, "Virtualizing reconfigurable hardware to provide scalability in cloud architectures," in *Proc. Reconfigurable Archit., Tools Appl.*, 2017, pp. 1–7.
[24] *Amazon FPGA Image (AFI) Management Tools*. [Online]. Available: https://github.com/aws/aw-fpga/blob/master/sdk/userspace/fpga_mgmt_tools
[25] E. Izenberg, "FPGA-enabled compute instances," U.S. Patent 14/986 330, Dec. 31, 2015.
[26] *Intel FPGA SDK for OpenCL. Kernel Design Concepts*. Accessed: 2018. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807516407.html
[27] A. Al-Aghbari and M. E. S. Elrabaa, "A platform for FPGA virtualization in clouds and data centers," *Microprocess. Microsyst.*, vol. 62, pp. 61–71, Oct. 2018.
[28] J. Weerasinghe, R. Polig, F. Abel, and C. Hagleitner, "Network-attached FPGAs for data center applications," in *Proc. Int. Conf. Field-Program. Technol. (FPT)*, Xi'an, China, 2016, pp. 36–43.
[29] (2010). *ARM AMBA AXI4-Stream Protocol Specifications*. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0051a/index.html
[30] W. Wang, M. Bolic, and J. Parri, "pvFPGA: Accessing an FPGA-based hardware accelerator in a paravirtualized environment," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synth. (CODES+ISSS)*, Montreal, QC, Canada, Sep. 2013, pp. 1–9.
[31] *The SDAccel Development Environment*. Accessed: 2018. [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html
[32] F. Haque, J. Michelson, and K. Khan, *The Art of Verification With VERA*, 1st ed. Verification Central, 2001. [Online]. Available: http://www.verificationcentral.com/product/the-art-of-verification-with-vera/
[33] M. Asiatici, N. George, K. Vipin, S. A. Fahmy, and P. Ienne, "Designing a virtual runtime for FPGA accelerators in the cloud," in *Proc. 26th Int. Conf. Field Program. Logic Appl. (FPL)*, Lausanne, Switzerland, Aug. 2016, pp. 1–2.
[34] 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.
[35] J. Zhang *et al.*, "The Feniks FPGA operating system for cloud computing," in *Proc. 8th Asia–Pacific Workshop Syst. (APSys)*, 2017, pp. 1–7. doi: 10.1145/3124680.3124743.
[36] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, "RIFFA 2.1: A reusable integration framework for FPGA accelerators," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 4, pp. 1–23, Sep. 2015. doi: 10.1145/2815631.
[37] K. Vipin and S. A. Fahmy, "DyRACT: A partial reconfiguration enabled accelerator and test platform," in *Proc. 24th Int. Conf. Field Program. Logic Appl. (FPL)*, Munich, Germany, Sep. 2014, pp. 1–7.
[38] M. Vesper, D. Koch, K. Vipin, and S. A. Fahmy, "JetStream: An open-source high-performance PCI Express 3 streaming library for FPGA-to-Host and FPGA-to-FPGA communication," in *Proc. 26th Int. Conf. Field Program. Logic Appl. (FPL)*, Lausanne, Switzerland, Aug. 2016, pp. 1–9.

AMRAN A. AL-AGHBARI received the B.Sc. degree in computer science from Sana'a University, Sana'a, Yemen, in 2004, and the M.Sc. degree in computer engineering from the King Fahd University of Petroleum and Minerals, in 2012, where he is currently pursuing the Ph.D. degree in computer science and engineering. He was a Lecturer with the Computer Science Department, Taiz University, Taiz, Yemen, from 2005 to 2009. His research interests include hardware/software co-design, hardware design languages, and virtualized reconfigurable computing.

MUHAMMAD E. S. ELRABAA received the M.A.Sc. and Ph.D. degrees in electrical and computer engineering from the University of Waterloo, Waterloo, ON, Canada, in 1991 and 1995, respectively. From 1995 to 1998, he was a Senior Component Designer with Intel Corporation, Portland, OR, USA. He is currently an Associate Professor with the Computer Engineering Department, King Fahd University of Petroleum and Minerals (KFUPM). He has authored or co-authored numerous papers and a book. He holds seven U.S. patents. His current research interests include reconfigurable computing, cloud-based custom computing machines, and systems-on-chip.