Enabling FPGAs in Hyperscale Data Centers

Jagath Weerasinghe, Francois Abel, Christoph Hagleitner
IBM Research - Zurich
Säumerstrasse 4
8803 Rüschlikon, Switzerland
Email: {wee,fab,hle}@zurich.ibm.com

Andreas Herkersdorf
Institute for Integrated Systems
Technical University of Munich,
Munich, Germany
Email: herkersdorf@tum.de

***Abstract*—FPGAs (Field Programmable Gate Arrays) are making their way into data centers (DCs) and are used to offload and accelerate specific services, but they are not yet available to cloud users. This puts the cloud deployment of compute intensive workloads at a disadvantage compared to on-site infrastructure installations, where the performance and energy-efficiency of FPGAs are increasingly exploited for application-specific accelerators and heterogeneous computing.**

The cloud is housed in DCs and DCs are based on ever shrinking servers. Today, we observe the emergence of hyperscale data centers, which are based on densely packaged servers. The shrinking form factors pose serious deployment challenges for the traditional PCIe-bus attached FPGAs. Therefore, a paradigm change is required for the CPU-FPGA and FPGA-FPGA interfaces. We propose an architecture that presents the FPGA as a standalone resource connected to the DC network.

This allows cloud vendors to offer an FPGA to users in a similar way as a standard server. Since existing infrastructure-as-a-service (IaaS) mechanisms are not suitable, we propose a new OpenStack (open source cloud computing software) service to integrate FPGAs in the cloud. This proposal is complemented by a framework that enables cloud users to combine multiple FPGAs into a programmable fabric. The proposed architecture and framework address the scalability problem that makes it difficult to provision large numbers of FPGAs. Together, they offer a new solution to process large and heterogeneous data sets in the cloud.

***Keywords*—FPGA; hyperscale data centers; cloud computing.**

I. Introduction

The use of FPGAs in application-specific processing and heterogeneous computing domains has been popular for 20 years or so. However, FPGAs have only just started to be used in cloud DCs for offloading and accelerating specific application workloads [1]. Even though FPGAs are not yet accessible to cloud users, they will make significant inroads into the cloud with more and more workloads being deployed in the cloud.

Enabling FPGAs on a large scale opens new opportunities for both cloud customers and cloud vendors. From the customers' perspective, FPGAs can be rented, used and released, similar to cloud infrastructure resources such as virtual machines (VMs) and storage. For example, IaaS users can rent FPGAs for education (e.g., university classes), research (e.g., building HPC systems) and testing (e.g., evaluation prior to deployment in real environments) purposes. From the Platform as a Service (PaaS) vendors' perspective, FPGAs can be used to offer acceleration as a service to the application developers on cloud platforms. For example, PaaS vendors can provide FPGA-accelerated application interfaces to PaaS users. A Software as a Service (SaaS) vendor can use FPGAs to provide acceleration as a service as well as to improve user experience. Acceleration of Bing web search service is such an example [1].

From the FPGA vendors' perspective, the cloud expands the FPGA user base and also opens new paths for them to market their own products. For example, new products can be placed on the cloud for users to try them out before actually purchasing them. In summary, the deployment of FPGAs in DCs will benefit both the users and the various cloud service providers and operators.

Meanwhile, the servers, which make up a cloud DC are continuously shrinking in terms of the form factor. This leads to the emergence of a new class of hyperscale data centers (HSDC) based on small and dense server packaging. The small form factor or dense packaging will enable to deploy a large number of FPGAs, exceeding by far the scaling capacity of the traditional PCIe-bus attachment.

To enable large-scale deployment of FPGAs in future HS-DCs, we advocate for a change of paradigm in the CPU-FPGA and FPGA-FPGA interfaces. We propose an architecture that sets the FPGA free from the CPU and its PCIe-bus by connecting the FPGA directly to the DC network as a standalone resource.

Cloud vendors can then provision these FPGA resources in a similar manner to servers. However, as existing server provisioning mechanisms are not suitable for this purpose, we propose a new resource provisioning service in OpenStack for integrating such standalone FPGAs in the cloud. Once such network-attached FPGAs become available on a large scale in DCs, they can be rented by cloud users. We further propose a framework for users to interconnect multiple FPGAs into a programmable fabric, and for the cloud vendor to deploy such a fabric in their infrastructure.

The proposed architecture and framework enable the provisioning of a large number of FPGAs for the cloud users. Users will be able to implement customized fabrics in a cost-effective, scalable and flexible manner in the cloud. These proposals offer new technical perspectives and solutions for processing large and heterogeneous data sets in the cloud.

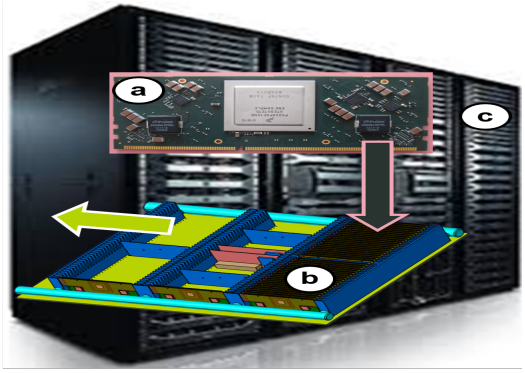

Fig. 1. Hyperscale Data Center: (a) Compute module with 24 GB DRAM (on the back). (b) 2U Rack chassis with 128 compute modules. (c) Four racks with 8K compute modules

II. Hyperscale Data Centers

The scaling of modern DCs has been fuelled by the continuous shrinking of the server node infrastructure. After the tower-, rack- and blade-server form factors, a new class of hyperscale servers (HSS) is emerging. In an HSS, the form factor is almost exclusively optimized for the performance-per-cost metric. This is achieved by increasing the density of CPUs per real estate and by sharing the cost of resources such as networking, storage, power supply, management and cooling.

At the time of writing, there are several HSSs on the market [2] [3] [4] and at the research stage [5] [6]. Among these, the HSS of DOME [5] has the objective of building the world's highest density and most energy efficient rack unit. In this paper, we refer to that specific type of HSS for supporting our discussion on hyperscale data centers (HSDC). Figure 1 shows the packaging concept of the HSS rack chassis (19" by 2U) proposed in [5]. In essence, this HSS is disaggregated into multiple node boards, each the size of a double-height dual in-line memory module (DIMM - 133mm x 55mm), which are densely plugged into a carrier base board. A node board with a CPU and its DRAM is called a compute module (Figure 1-(a)). Similarly, a node board with solid-state disks (SSD) is called a storage module, and a node board with a Ethernet switch is referred to as a networking module. The use of a homogeneous form factor is a significant contributor to the overall cost minimization of an HSS.

The base board is a passive board that provides system management, 10 GbE networking between the node boards and multiple 40 GbE uplinks. This ultra-dense packaging is combined with an innovative cooling system that enables the integration of as many as 128 compute modules into a 2U chassis (Figure 1-(b)). For a 19" rack with 16 such chassis, this amounts to 2K compute modules and $\sim$50 TB of DRAM.

III. Prior Art

There are a few previous attempts to enable FPGAs in the cloud. Chen et al. [7] and Byma et al. [8] proposed frameworks to integrate virtualized FPGAs into the cloud using the OpenStack infrastructure manager. In both cases, FPGAs are virtualized by partitioning them to multiple slots, where each slot or virtual FPGA is a partially reconfigurable region in a physical FPGA which is attached over a PCIe-bus.

In [7], a virtual FPGA model is present in each virtual machine (VM) and acts as the communication channel between the applications running in the VM and the virtual FPGA. The commands and data communicated by the virtual FPGA model are transferred to the virtual FPGA by the hypervisor. There are a few drawbacks in this framework particularly from the perspective of a cloud-based framework. First, users can not deploy their own designs in the FPGAs, instead a limited set of applications offered by the cloud vendor has to be used. Second, if a user needs to deploy an application using several virtual FPGAs connected together, the data have to be copied back and forth through the VM and hypervisor stack to feed the next FPGA. Third, VM migration disrupts the use of the virtual FPGA because the physical FPGA is tightly coupled to the hypervisor.

In contrast to [7], the framework proposed by [8] allows users to deploy their own application in the FPGAs and allows those applications to be accessed over the Ethernet network. In addition to that, [8] has shown that the OpenStack *Glance* image service can be used for managing FPGA bit streams, which is an important factor when integrating FPGAs into OpenStack. However, from the perspective of a cloud deployment, also this framework has a few drawbacks. First, as network, a plain Ethernet connection is offered to the virtual FPGAs which limits the flexibility of the applications that can run on FPGAs. Second, even though multiple virtual FPGAs are enabled in a single physical FPGA, an isolation method, such as VLAN or overlay virtual network (OVN) for multi-tenant deployments is not supported, which is indispensable in deploying applications in the cloud.

Catapult [1] is a highly customized, application-specific FPGA-based reconfigurable fabric designed to accelerate page-ranking algorithms in the Bing web search engine. It is a good example to show the potential of FPGAs on a large scale in cloud DCs. Authors claim that compared with a software-only approach, Catapult can achieve 95% improvement in ranking throughput for a fixed latency. Even though it has shown good results, as a system deployed in a DC it has a few drawbacks. In Catapult, similarly to the above-mentioned systems, FPGAs are PCIe-attached. But for scaling, these PCIe-attached FPGAs are connected by a dedicated serial network, which breaks the homogeneity of the DC network and increases the management overhead. Even though maintaining a dedicated network is a significant management overhead, it can be traded off for the good performance achieved by Catapult. However, in the case of general-purpose cloud DCs, maintaining such a customized infrastructure is not acceptable because the FPGAs are used in diverse kinds of applications at different scales similarly to other DC resources such as servers and storage.

All these systems deploy FPGAs tightly coupled to a sever over the PCIe bus. In [1] and [7], FPGAs are accessed through the PCIe bus, whereas [8] uses a plain Ethernet connection.

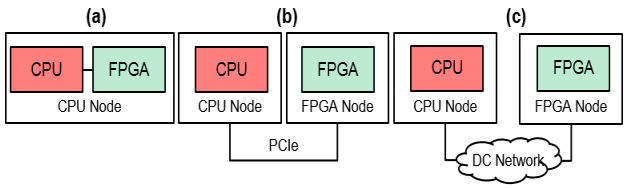

Fig. 2. Options for Attaching an FPGA to a CPU

In [1], FPGAs are chained in a dedicated network for scaling. In contrast to those systems, the focus of our proposal is to consider the FPGA as a DC network-connected standalone resource with compute, memory and networking capabilities. In the context of an HSDC, the FPGA resource is therefore considered as a hyperscale server-class computer.

IV. FPGA Attachment Options

The miniaturization of the DC servers is a game-changing requirement that will transform the traditional way of instantiating and operating an FPGA in a DC infrastructure. We consider three approaches for setting up a large number of FPGAs into an HSDC.

One option is to incorporate the FPGA onto the same board as the CPU when a tight or coherent memory coupling between the two devices is desired (Figure 2-(a)). We do not expect such a close coupling to be generalized outside the scope of very specific applications, such as web searching or text-analytics processing [9]. First, it breaks the homogeneity of the compute module in an environment where server homogeneity is sought to reduce the management overhead and provide flexibility across compatible hardware platforms. Second, in large DCs, failed resources can be kept in place for months and years without being repaired or replaced, in what is often referred to as a fail-in-place strategy. Therefore, an FPGA will become unusable and its resources wasted if its host CPU fails. Third, the footprint of the FPGA takes a significant real estate away from the compute module –the layout of a large FPGA on a printed circuit board is somehow equivalent to the footprint of a DDR3 memory channel, i.e. 8-16GB–, which may require the size of the module to be increased (e.g., by doubling the height of the standard node board from 2U to 4U). Finally, the power consumption and power dissipation of such a duo may exceed the design capacity of a standard node board.

The second and by far the most popular option in use today is to implement the FPGA on a daughter-card and communicate with the CPU over a high-speed point-to-point interconnect such as the PCIe-bus (Figure 2-(b)). This path provides a better balance of power and physical space and is already put to use by FPGAs [1] as well as graphics processing units (GPU) in current DCs. However, this type of interface comes with the following two drawbacks when used in a DC. First, the use of the FPGA(s) is tightly bonded to the workload of the CPU, and the fewer the PCIe-buses per CPU, the higher is the chance of under-provisioning the FPGA(s), and vice-versa. Catapult [1] uses one PCIe-attached FPGA per CPU and solves this inelastic issue by deploying a secondary inter-FPGA network at the price of additional cost, increased cabling and management complexity. Second, server applications are often migrated within DCs. The PCIe-attached FPGA(s) affected must then be detached from the bus before being migrated to a destination where an identical number and type of FPGAs must exist, thus hindering the entire migration process. Finally, despite the wide use of this attachment model in high-performance computing, we do not believe that is a way forward for the deployment of FPGAs in the cloud because it confines this type of programmable technology to the role of coarse accelerator in the service of a traditional CPU-centric platform.

The third and preferred method for deploying FPGAs in an HSDC is to set the FPGA free from the traditional CPU-FPGA attachment by hooking up the FPGA directly to the HSDC network (Figure 2-(c)). The main implication of this scheme is that the FPGA must be turned into a standalone appliance capable of communicating with a host CPU over the network of the DC. From a practical point of view, and with respect to the HSDC concept of section II, this is an FPGA module equipped with an FPGA, some optional local memory and a network controller interface (NIC). Joining a NIC to an FPGA enables that FPGA to communicate with other DC resources, such as servers, disks, I/O and other FPGA modules. Multiple such FPGA modules can then be deployed in the HSDC independently of the number of CPUs, thus overcoming the limitations of the two previous options.

The networking layer of such an FPGA module can be implemented with a discrete or an integrated NIC. A discrete NIC (e.g., dual 10 GbE NIC) is a sizable application-specific integrated circuit (ASIC) typically featuring 500+ pins, 400+ mm^2 of packaging, and 5 to 15 W of power consumption. The footprint and power consumption of such an ASIC do not favour a shared-board implementation with the FPGA (see above discussion on sharing board space between an FPGA and a CPU). Inserting a discrete component also adds a point of failure in the system. Integrating the NIC into the reconfigurable fabric of the FPGA alleviates these issues. Furthermore, it provides the agility to implement a specific protocol stack on demand, such as Virtual Extensible LAN (VxLAN), Internet Protocol version 4 (IPV4), version 6 (IPv6), Transmission Control Protocol (TCP), User Datagram Protocol (UDP) or Remote Direct Memory Access (RDMA) over Converged Ethernet (RoCE). Finally, the configurable implementation can also adapt to emerging new protocols, such as Generic Network Virtualization Encapsulation (Geneve) and Network Virtualization Overlays (NVO3).

In summary, we advocate a direct attachment of the FPGA to the DC network by means of an integrated NIC, and refer to such a standalone FPGA as a network-attached FPGA. The combination of such network-attached FPGAs with emerging software-defined networking (SDN) technologies brings new technical perspectives and market value propositions such as building large and programmable fabrics of FPGAs on the

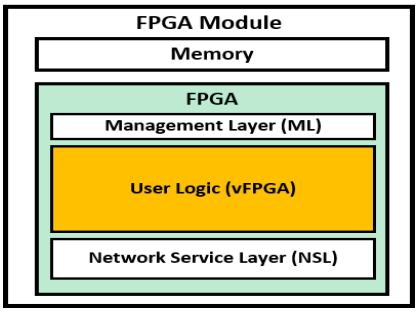

Fig. 3. High-level Architecture of the Network-attached FPGA Module

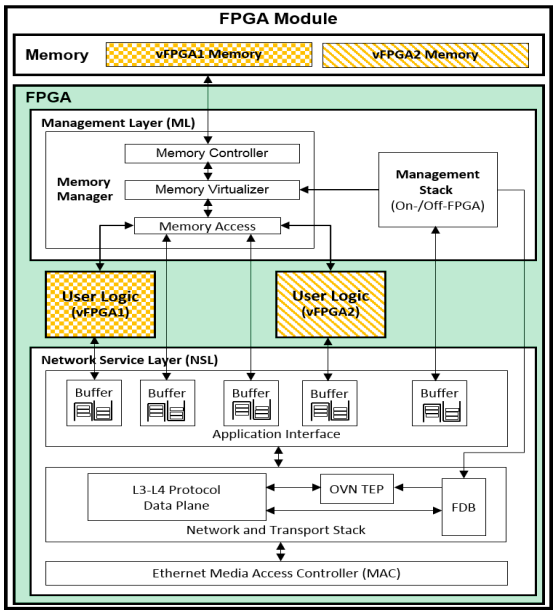

Fig. 4. Low-level Architecture of the Network-attached FPGA Module

cloud.

V. System Architecture

In this section, we propose and describe *A*) the architecture of such a network-attached FPGA, *B*) the way it is integrated into a cloud environment, and *C*) how it can be deployed and used on a large scale.

A. *Network-attached FPGA Architecture*

The high-level architecture of the proposed network-attached FPGA concept is shown in Figure 3. It contains an FPGA and an optional off-chip memory. The FPGA is split into three main parts: *i*) a user logic part used for implementing customized applications, *ii*) a network service layer (NSL), which connects with the DC network, and *iii*) a management layer (ML) to run resource-management tasks.

In the context of an HSDC, the FPGA concept of Figure 3 is matched to the double-height DIMM form factor defined in section II and is therefore referred to as an FPGA module. The architecture of such a network-attached FPGA module is now explained in detail with reference to Figure 4.

*1) **User Logic (vFPGA)**:* Multiple user applications can be hosted on a single physical FPGA (pFPGA), somehow similar to multiple VMs running on the same hypervisor. Each user gets a partition of the entire user logic and uses it to implement its applications. This partitioning is achieved by a feature called partial reconfiguration, a technology used to dynamically reconfigure a region of the FPGA while other regions are running untouched[1]. We refer to such a partition of user logic as a virtual FPGA (vFPGA), and it is depicted in Figure 4 as vFPGA1 and vFPGA2[2]. For the sake of simplicity, in this discussion we assume there is only one vFPGA in the user logic. A vFPGA is assigned an ID (vFPGAID), an IP address, a MAC address and a tenant ID. The vFPGA connects to the DC network through the NSL, and can therefore communicate with other vFPGAs. A vFPGA can also have off-chip local memory assigned to it.

*2) **Network Service Layer (NSL)**:* The NSL is a HW implementation of the physical, data link, network and transport layers (L1-L4) used in a typical protocol layered architecture. These layers are mapped into the following three components: *i*) an Ethernet media access (MAC) controller, *ii*) a network and transport stack, and *iii*) an application interface.

*a) **Ethernet MAC**:* The Ethernet MAC implements the data link layer of the Ethernet standard. The MAC performs functions such as frame delineation, cyclic redundancy check (CRC), virtual LAN extraction and collection of statistics. Normally, a MAC connects to an external physical layer device (PHY) whose task is to physically move the bits over the media by means of encoding/decoding and serialization/deserialization techniques. However, because of the dense packing of an HSDC, the modules plugged on the same chassis base board are all located within a short reach and do not require such a PHY to communicate with each other.

Therefore, the MAC implements the networking interface of the FPGA module by connecting directly to the high-speed transceivers provided in every mid- and high-end FPGA. These integrated transceivers operate at line rates up to 16 Gb/s and they commonly support the Ethernet standards 10 GBASE-KR (10 Gb/s) and 40 GBASE-KR4 (40 Gb/s) which we seek for interconnecting our modules over a distance up to 1 meter of copper printed circuit board and two connectors. This PHY removal is a key contributor in the overall power, latency, cost and area savings.

*b) **Network and Transport Stack**:* The network and transport stack provides a HW implementation of L3-L4 protocols. Applications running on a cloud HW infrastructure are inherently diverse. These applications impose different communication requirements on the infrastructure. For example, one system may require a reliable, stream-based connection such as TCP for inter-application communication, whereas another system may need an unreliable, message-oriented communication, such as UDP. For the applications where la-

[1]This partial reconfiguration feature is not further discussed as it exceeds the scope of this paper

[2]Note that in the figure, vFPGA1 and vFPGA2 are not proportional in size to the NSL and the ML

tency is critical, RoCE might be preferred. Having this network and transport stack implemented in HW within the FPGA provides low-latency and enables to instantiate these protocols on demand. Again, we leverage partial reconfiguration feature of the FPGA to achieve such a flexibility.

Usually, a network protocol stack contains a control plane and a data plane. The control plane learns how to forward packets, whereas the data plane performs the actual packet forwarding based on the rules learnt by the control plane. Usually, these two planes sit close to each other in a network stack, with the control plane distributed over the network. With the emergence of SDN, we observe that these planes are getting separated from each other. In the FPGA, it is important to use as few resources as possible for the NSL in order to leave more space for the user logic. To minimize the complexity of the stack, inspired by the SDN concepts, we decouple the control plane from the HW implementation of the data plane and place it in software.

The vFPGAs must be securely isolated in multi-tenant environments. For this isolation, it is important to use widely used techniques such as VLANs or OVNs in order to coexist with other infrastructure resources. Therefore, we implement a tunnel endpoint (TEP) of an OVN in the network and transport stack. The TEP implemented in FPGA hardware also provides an acceleration, as software-based TEPs degrade both the network and CPU performance significantly [10].

The forwarding data base (FDB) sitting in the network and transport stack contains the information on established connections belonging to connection-oriented protocols and the information on allowed packets from connection-less protocols. This information includes mac addresses, IP addresses, OVN IDs and port numbers belonging to source and destination vFPGAs. The control plane running in a centralized network management software feeds this information to the FDB through the ML.

*c) **Application Interface**:* The application interface comprises FIFO-based connection interfaces resembling socket buffers in a TCP or UDP connection. The vFPGA reads from and writes to these FIFOs to communicate with other vFPGAs. One or more FIFO interfaces can be assigned to a single vFPGA.

*3) **Management Layer (ML)**:* The management layer contains a memory manager and a management stack. The memory manager enables access to memory assigned to vFPGAs and the management stack enables the vFPGAs to be remotely managed by a centralized management software.

*a) **Memory Manager**:* The memory manager contains a memory controller and a virtualization layer. The memory controller provides the interface for accessing memory from the vFPGAs. The virtualization layer allows the physical memory to be partitioned and shared between different vFPGAs in the same device. This layer is configured through the management stack according to the vFPGA memory requirements. It uses the vFPGAID to calculate the offset when accessing the physical memory that belongs to a particular vFPGA.

*b) **Management Stack**:* The management stack runs a set of agents to enable the centralized resource-management software to manage the FPGA remotely. The agents include functions such as device registration, network and memory configuration, FPGA reconfiguration, and a service to make the FPGA nodes discoverable. The management stack may run on an embedded operating system in a soft core processor or preferably in a hard core processor, like the processing system in a Xilinx FPGA Zynq device. The network connection of the embedded OS is then shared with the HW network stack of the NSL to reduce the number of physical network connections to the FPGA module.

B. Cloud integration

Cloud integration is the process of making the above-mentioned vFPGAs available in the cloud so that users can rent them. In this section, we present a framework for integrating FPGAs in the cloud that consists of a new accelerator service for OpenStack, a way to integrate FPGAs into OpenStack, a way to provision FPGAs on the cloud, and a way for the user to rent an FPGA on the cloud.

*1) **Accelerator Service for OpenStack**:* We propose a new service for OpenStack to enable network-attached FPGAs. In previous research, FPGAs [8] [7] and GPUs [11] have been integrated into the cloud by using the *Nova* compute service in OpenStack. In those cases, heterogeneous devices are PCIe-attached and are usually requested as an option with virtual machines or as a single appliance, which requires a few simple operations to make the device ready for use.

In our deployment, in contrast, standalone FPGAs are requested independent of a host because we want to consider them as a new class of compute resource. Therefore, similar to *Nova*, *Cinder* and *Neutron* in OpenStack, which translate high-level service API calls into device-specific commands for compute, storage and network resources, we propose the accelerator service shown in Figure 5, to integrate and provision FPGAs in the cloud. In the figure, the parts in red show the new extensions we propose for OpenStack. To setup network connections with the standalone FPGAs we need to carry out management tasks. For that, we use an SDN stack connected to the *Neutron* network service, and we call it the network manager. Here we explain the high-level functionality of the accelerator-service and the network-manager components.

Accelerator Service: The accelerator service comprises an API front end, a scheduler, a queue, a data base of FPGA resources (DB), and a worker. The API front end receives the accelerator service calls from the users through the OpenStack dashboard or through a command line interface, and dispatches them to the relevant components in the accelerator service. The DB contains the information on pFPGA resources. The scheduler matches the user-requested vFPGA to the user logic of a pFPGA by searching the information in the DB, and forwards the result to the worker. The worker executes four main tasks: *i*) registration of FPGA nodes in the DB; *ii*) retrieving vFPGA bit streams from the *Swift* object store; *iii*) forwarding service calls to FPGA plug-ins, and *iv*) forwarding

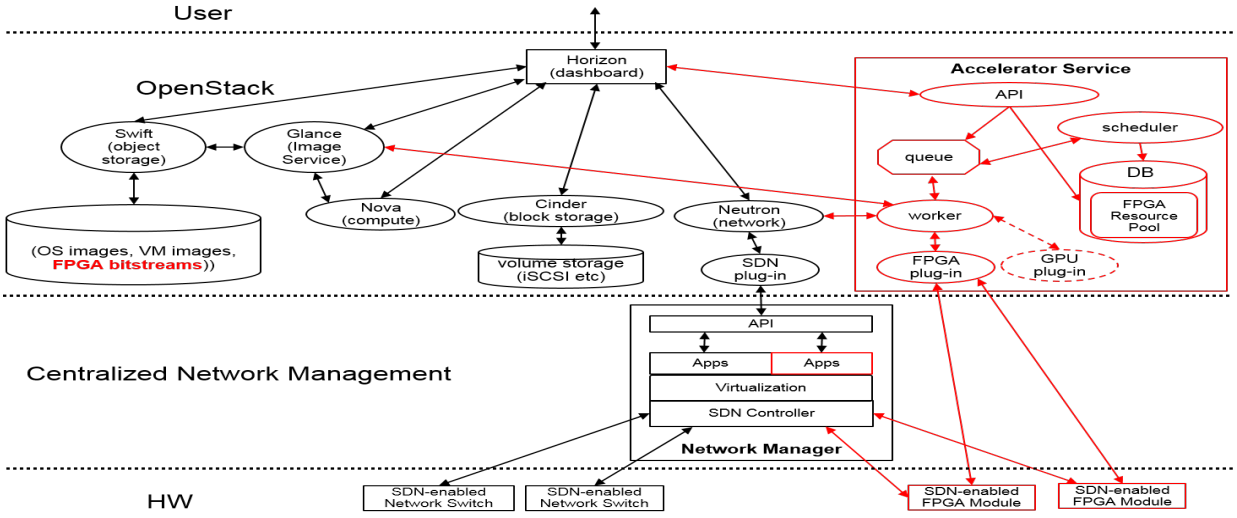

Fig. 5. OpenStack Architecture with Network-attached FPGAs

network management tasks to the network manager through the *Neutron* service. The queue is just there to pass service calls between the API front end, the scheduler and the worker. The FPGA plug-in translates the generic service calls received from the worker into device-specific commands and forwards them to the relevant FPGA devices. We foresee the need for one specific plug-in per FPGA vendor to be hooked to the worker. Other heterogeneous devices like GPUs and DSPs will be hooked to the worker in a similar manner.

Network Manager: The network manager is connected to the OpenStack *Neutron* service through a plug-in. The network manager has an API front end, a set of applications, a network topology discovery service, a virtualization layer, and an SDN controller. The API front end receives network service calls from the accelerator-worker through the *Neutron* and exposes applications running in the network manager. These applications include connection management, security and service level agreements (shown in red in the network manager in Figure 5). The virtualization layer provides a simplified view of the overall DC network, including FPGA devices, to the above applications. The SDN controller configures both the FPGAs and network switches according to the commands received by the applications through the virtualization layer.

2) ***Integrating FPGAs into OpenStack****:* In this sub section, the process of integrating FPGAs into OpenStack is outlined. The IaaS vendor executes this process as explained below.

When the IaaS vendor powers up an FPGA module, the ML of the FPGA starts up with a pre-configured IP address. This IP address is called the management IP. The accelerator service and the network manager use this management IP to communicate with the ML for executing management tasks. Second, the network-attached FPGA module is registered in the accelerator-DB in the OpenStack accelerator service. This is achieved by triggering the registration process after entering the management IP into the accelerator service. Then the accelerator service acquires the FPGA module information automatically from the ML over the network and stores them in the FPGA resource pool in the accelerator-DB. Third, a few special files [3] needed for vFPGA bitstream generation are uploaded to the OpenStack *Swift* object store.

3) ***Provisioning an FPGA on the Cloud****:* From the IaaS vendors' perspective, let's now look at the process of provisioning a single vFPGA. When a request for renting a vFPGA arrives, the accelerator-scheduler searches the FPGA pool to find a user logic resource that matches the vFPGA request. Once matched, the tenant ID and an IP address are configured for the vFPGA in the associated pFPGA. After that, the vFPGA is offered to the user with a few special files which are used to generate a bitstream for user application.

4) ***Renting an FPGA on the Cloud****:* From the user's perspective, the process of renting a single vFPGA on the cloud and configuring a bitstream to it is as follows. First, the user specifies the resources that it wants to rent by using a GUI provided by the IaaS vendor. This includes FPGA-internal resources, such as logic cells, DSP slices and Block RAM as well as module resources, such as DC network bandwidth and memory capacity. The IaaS vendor uses this specification to provision a vFPGA as explained above.

Upon success, a reference to the provisioned vFPGA is returned to the user with a vFPGAID, an IP address and the files needed to compile a design for that vFPGA. Second, the user compiles it's design to a bitstream and uploads it to the OpenStack *Swift* object store through the *Glance* image service. Finally, the user associates the uploaded bitstream with the returned vFPGAID and requests the accelerator service to boot that vFPGA. At the successful conclusion of the renting process, the vFPGA and its associated memory are accessible over the DC network.

[3]Users need these files to generate a bitstream for the vFPGA

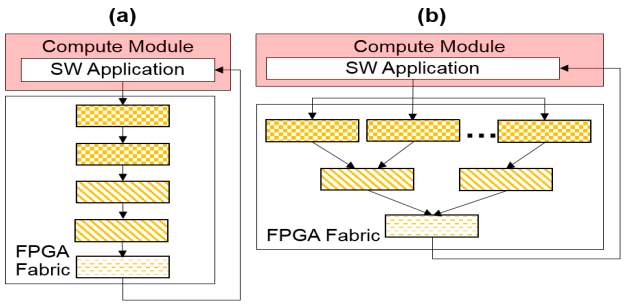

Fig. 6. Two Examples of FPGA Fabrics

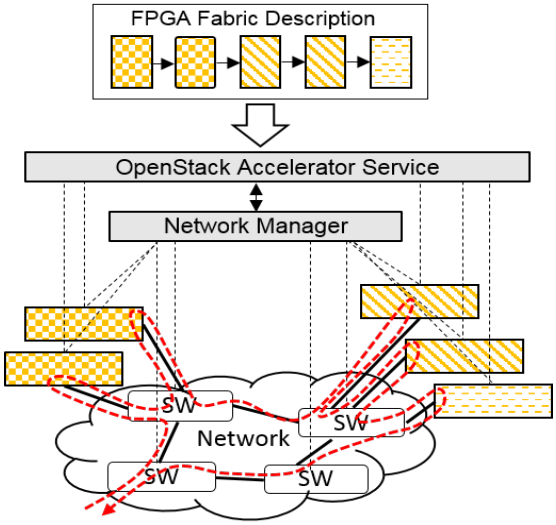

Fig. 7. FPGA Fabric Deployment; SW: Network Switch

C. *Fabric of FPGAs on the Cloud*

Motivated by the success of large-scale SW-based distributed applications such as those based on MapReduce and deep learning [12], we want to give the users a possibility to distribute their applications on a large number of FPGAs. This sub section describes a framework for interconnecting such a large number of FPGAs in the cloud that offers the potential for FPGAs to be used in large-scale distributed applications.

We refer to a multiple number of FPGAs connected in a particular topology as an FPGA fabric. When the interconnects of this fabric are reconfigurable, we refer to it as a programmable fabric. Users can define their programmable fabrics of vFPGAs on the cloud and rent them using the proposed framework. Figure 6 shows such two fabrics in which vFPGAs with different sizes are shown in different patterns. These two fabrics are used to build two different types of applications. As an example, a fabric of FPGAs arranged in a pipeline, shown in Figure 6-(a), is used in Catapult [1] for accelerating page-ranking algorithms, which we discussed in prior art. Figure 6-(b)) shows a high-level view of a fabric that can be used for mapping and reducing type of operations.

*1) **Renting an FPGA Fabric on the Cloud**:* The renting and provisioning steps of such a fabric in the cloud are as follows. First, user decides on the required number of vFPGAs and customizes them as mentioned above in the case of a single vFPGA. Second, the user defines its fabric topology by connecting those customized vFPGAs on a GUI or with a script. We call this fabric a vFPGA Fabric (vFF). In vFF, the number of network connections between two vFPGAs can be selected. If a network connection is required between a vFPGA and the SW applications that uses the vFF (explained in the next sub section), it is also configured in this step. Third, the user rents the defined vFF from the IaaS vendor. At this step, the user-defined fabric description is passed to the OpenStack accelerator service. Then, similar to a single vFPGA explained earlier, the accelerator service matches the vFF to the hardware infrastructure as shown in Figure 7. In addition to the steps followed when matching a single vFPGA, the scheduler considers the proximity of vFPGAs and optimal resource utilization when matching a vFF to the hardware infrastructure. After that, the accelerator service requests the network manager to configure the NSL of assoicated pFPGAs and intermediate network switches to form the fabric in HW infrastructure. Fourth, the user associates a bitstream with each vFPGA of the vFF and requests to boot the fabric. Finally, on successful provisioning, an ID representing the fabric (vFFID) is returned to the users that is used in the programming phase to access the vFF.

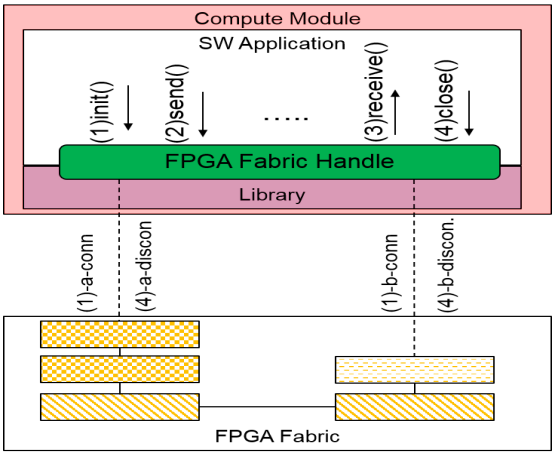

Fig. 8. FPGA Fabric Programming Model

*2) **Using an FPGA Fabric from SW Applications**:* The way a vFF is used from a SW application is explained here. We consider the pipeline-based vFF shown in Figure 6-(a) as an example and show how it can be used from a SW application. We assume this fabric runs an application based on data-flow computing. The text-analytics acceleration engine explained in [9] is an example of such an application. Also, we assume that the L4 protocol used is a connection-oriented protocol such as TCP.

To make the applications agnostic to the network protocols and to facilitate the programming, we propose a library and an API to use both the vFPGAs and vFFs. The vFFID returned at the end of the fabric-deployment phase is used

to access the vFF from the SW applications by means of the vFFID. Below are the steps for accessing a vFF. First, the vFF is initialized from the SW application. This initiates a connection for sending data to the vFF as shown by (1)-a-conn in Figure 8 . The immediate SDN-enabled switch triggers this connection-request packet and forwards it to the network manager. On behalf of the first vFPGA in the pipeline, the network manager establishes the connection and updates the relevant FDB entries in the associated pFPGA. For receiving data, the library starts a local listener and tells the network manager to initiate a connection on behalf of the last vFPGA in the pipeline ((1)-b-conn). Then, the SW application can start sending data and receiving the result by calling send() and receive() on the vFFH, respectively. If configured by the user at the vFF definition stage, connections are created for sending back intermediate results from the vFPGA to the SW application. When close() is called on the vFFH, the connections established are closed detaching the fabric from the SW application. The connections for accessing memory associated with each vFPGA are also established in a similar manner through the network manager in the fabric initialization phase. The SW applications can write to and read from the memory using the vFPGAID.

VI. Initial Results

A. *Resource Estimation*

This section preestimates the FPGA resource utilization for the NSL and the ML. According to the commercial implementations, an UDP and a TCP engine consume approximately 5K [13] and 10K [14] lookup tables (LUTs), respectively. A memory controller requires around 4K LUTs [15]. Assuming that the management stack in the ML is implemented in a hard core processor, we estimate the total resource utilization for both the NSL and the ML to be approximately 30K LUTs, which accounts for 15 to 20% of the total LUT resources available in a modern FPGA device such as the Xilinx 7 series. This resource utilization is comparable to previous attempts of enabling FPGAs in the cloud based on PCIe [1] [8], which use 20-25% of FPGA resources for communication and management tasks. However, as we already discussed, the network-attached FPGA we propose alleviate most of the limitations posed by the PCIe-bus enabling large scale deployments in the DCs.

B. *Scaling Perspectives*

As explained earlier, the network-attached FPGA module enables large scale deployment of FPGAs in DCs similarly to compute modules. Table I shows the single precision floating-point compute performance of a full rack of resources in an HSDC built using FreeScale T4240-based [16] compute modules [5] and Xilinx Zynq 7100 SoC-based [17] FPGA modules. Such a full rack of FPGAs can achieve close to 1000 TFLOPS and provides the user with the impressive number of 71 million configurable logic blocks (CLBs). Here, we assume that only 32U of rack space is used, out of 42U, for deploying above resources.

TABLE I
Compute Performance Per Rack

Per Rack	Hyperscale Server	Hyperscale FPGA
Modules	2048	2048
Cores	24586	4096 (ARM)
CLBs (10^6)	–	71
Memory (TB)	49	49
TFLOPS	442	958

VII. Conclusion

As FPGAs can be reconfigured for specific workloads, enabling them in data centers opens up new business opportunities and technical perspectives for both cloud vendors and customers. In this paper, we present system architectures that enables large-scale FPGA deployments in hyperscale data centers. We envision that the programmable fabrics of FPGAs we propose using standalone network-attached FPGAs will revolutionize the way heterogeneous data sets are processed in the cloud in future data centers.

References

[1] A. Putnam *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 13–24.

[2] Hewlett-Packard, "HP Moonshot: An accelerator for hyperscale workloads," 2013. [Online]. Available: www.hp.com

[3] SeaMicro, "Seamicro SM15000 fabric compute systems." [Online]. Available: http://www.seamicro.com/

[4] Dell, "Dell poweredge C5220 microserver." [Online]. Available: http://www.dell.com/

[5] R. Luijten and A. Doering, "The DOME embedded 64 bit microserver demonstrator," in *2013 International Conference on IC Design Technology (ICICDT)*, May 2013, pp. 203–206.

[6] Hewlett Packard, "The machine: A new kind of computer." [Online]. Available: http://www.hpl.hp.com/

[7] F. Chen *et al.*, "Enabling FPGAs in the cloud," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, ser. CF '14. New York, NY, USA: ACM, 2014, pp. 3:1–3:10.

[8] S. Byma *et al.*, "FPGAs in the cloud: Booting virtualized hardware accelerators with openstack," in *Proceedings of the 2014 IEEE 22Nd International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '14, 2014, pp. 109–116.

[9] R. Polig *et al.*, "Giving text analytics a boost," *IEEE Micro*, vol. 34, no. 4, pp. 6–14, July 2014.

[10] J. Weerasinghe and F. Abel, "On the cost of tunnel endpoint processing in overlay virtual networks," in *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*, Dec 2014, pp. 756–761.

[11] S. Crago *et al.*, "Heterogeneous cloud computing," in *2011 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2011, pp. 378–385.

[12] J. Dean *et al.*, "Large scale distributed deep networks," in *Neural Information Processing Systems, NIPS 2012*.

[13] Xilinx, "UDPIP - Hardware UDP/IP Stack Core." [Online]. Available: http://www.xilinx.com

[14] Mle, "TCP/UDP/IP Network Protocol Accelerator." [Online]. Available: http://www.missinglinkelectronics.com

[15] G. Kalokerinos *et al.*, "FPGA implementation of a configurable cache/scratchpad memory with virtualized user-level RDMA capability," in *SAMOS '09. International Symposium on*, July 2009, pp. 149–156.

[16] FreeScale, "T4240 product brief," Oct 2014. [Online]. Available: www.freescale.com

[17] Xilinx, "DSP solution." [Online]. Available: http://www.xilinx.com/products/technology/dsp.html