Accepted for FPL 2019

System architecture for network-attached FPGAs in the Cloud using partial reconfiguration

Burkhard Ringlein*†, Francois Abel†, Alexander Ditter*, Beat Weiss†,
Christoph Hagleitner†, and Dietmar Fey*

†*IBM Research – Zurich*
Rüschlikon, Switzerland
{ngl, fab, wei, hle}@zurich.ibm.com

**Chair of Computer Architecture*
Friedrich-Alexander University
Erlangen-Nürnberg, Germany
{burkhard.ringlein, alexander.ditter, dietmar.fey}@fau.de

***Abstract*—Emerging applications such as deep neural networks, bioinformatics or video encoding impose a high computing pressure on the Cloud. Reconfigurable technologies like Field-Programmable Gate Arrays (FPGAs) can handle such compute-intensive workloads in an efficient and performant way.**
To seamlessly incorporate FPGAs into existing Cloud environments and leverage their full power efficiency, FPGAs should be directly attached to the data center network and operate independent of power-hungry CPUs. This raises new questions about resource management, application deployment and network integrity.
We present a system architecture for managing a large number of network-attached FPGAs in an efficient, flexible and scalable way. To ensure the integrity of the infrastructure, we use partial reconfiguration to separate the non-privileged user logic from the privileged system logic. To create a really scalable and agile cloud service, the management of all resources builds on the Representational State Transfer (REST) concept.

***Index Terms*—cloud computing, network-attached FPGA, stand-alone FPGA, partial reconfiguration, data centers**

I. Introduction

The growth of computing infrastructures has long been driven by Moore's law and the use of homogeneous CPU-centric platforms. These traditional systems are suffering due to extreme power density on the chips. Therefore, modern supercomputers and Cloud Data Centers (DC) take a new approach by exploiting high-performance and low-power heterogeneous devices. Field-Programmable Gate Arrays (FPGA) are such heterogeneous devices. Their energy efficiency and low latency have the potential to drastically improve the power density of compute nodes while also delivering results faster. Consequently, more and more FPGAs are making their way into DCs where they are used as accelerators to boost the computing power and overall power efficiency of individual server nodes.

Two cloud architectures have recently proposed to turn FPGA resources into independent cloud computer nodes by enabling the FPGAs to communicate directly at DC scale [1][2].

In [1], the FPGA is placed as a network-side "bump-in-the-wire" between the servers' Network Interface Controller (NIC) and the Ethernet network switches. The hyperscale infrastructure we proposed in [2] goes one step further by disaggregating the FPGA accelerator from the server and by turning the FPGA into a stand-alone computing resource. Such network-attached FPGAs can be deployed at large scale and independently of the number of CPU servers in the DC. The network attachment allows them to seamlessly connect with each other as well as with more than one CPU. The resulting disaggregated heterogeneous computing infrastructure is capable to dynamically adapt to the scale of any workload.

Meanwhile, large-scale applications ranging from business analytics to scientific simulations have started to scale out using distributed frameworks such as Hadoop, Spark, and Tensorflow. These frameworks will consume an immense number of FPGAs, which must be provided and managed in an efficient and flexible way.

We propose an architecture for acquiring, distributing, configuring and operating stand-alone network-attached FPGAs at a large scale in DC infrastructures. We start by defining a list of major requirements for building such a scalable deployment by investigating the constraints expressed by the DC provider and the FPGA users. Based on this analysis, we propose an architecture with three tiers covering all levels from the cores implementing the user application to the DC-management.

Stand-alone network-attached FPGAs come with specific management demands such as network setup, creation of user subnets, network routing and the distribution of the configuration bitfiles. We deliver these services via the network and show how they seamlessly integrate into the OpenStack DC management software.

We enforce the use of partial reconfiguration to physically isolate the system management functions from the user logic within the network-attached FPGA. *Partial Reconfiguration (PR)* is the ability to dynamically modify blocks of the FPGA logic while the remaining logic continues to operate without interruption [3]. This approach protects the integrity of the DC network by creating a separation between privileged and non-privileged user logic functions.

The rest of this paper is structured as follows: The next section discusses related research in this field. After that, we consider the constraints of a system architecture for a hyperscale FPGA service and present our implementation to realize such a service. Finally, we evaluate our implementation

©2019 IEEE Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribu-tion to servers or lists, or reuse of any copyrighted component of this work in other works.
This is the accepted version of the article published by IEEE: DOI 10.1109/FPL.2019.00054

and draw some conclusions.

II. Related Work

A well-known project for incorporating FPGAs in data-center-level applications is Microsoft's *Catapult* project. In 2014, *Putnam et al.* used PCIe-attached FPGAs in DC servers to accelerate the ranking engine of Bing. Their results showed an almost twofold increase of the throughput with comparable latency while consuming only 10% more energy [4]. The logic in the FPGA is split into an infrastructure part (*Shell*) and an application logic (*Role*) to increase the reuse of logic by the algorithm [1]. Their application was completely ported to Verilog.

In May 2018, Microsoft announced *Project Brainwave*, which builds on their Catapult Project and is expected to deliver real-time Artificial Intelligence (AI) analysis in the future [5][6].

The Catapult Fabric deploys FPGAs in a DC environment but they are not provided to users as a "cloud service". This is, for example, done by Amazon with their F1 Instances [7] or by IBM's SuperVessel Cloud [8]. Amazon offers FPGAs as part of its AWS cloud service and marketplace. As explained in the AWS EC2 FPGA Development Kit [9][10], FPGAs are attached via PCIe to a Virtual Machine (VM) and the logic is also split into a Shell, which contains among others the logic for PCIe, DRAM, and DMA, and a custom logic for the actual application. The user submits and uploads her or his application as a design-checkpoint (i.e. the sources must be included) to AWS. The AWS platform then merges it with the Shell logic and generates the FPGA image (i.e. bitstream). In contrast, SuperVessel brings FPGAs into an OpenStack-based Cloud by attaching them to POWER8 processors via IBMs CAPI (Coherent Accelerator Processor Interface) [11].

A joint project between IBM and Microsoft was performed by *Chen et al.* in 2014 [12] to analyse the impediments of using FPGAs as shareable resource in the Cloud. Their concept revolves around an FPGA board attached to a host CPU via PCIe, but in contrast to Aamazon's F1, it is possible to run multiple user kernels on one FPGA simultaneously using PR.

There are several projects from the Electronics Research Group of the University of Toronto [13] [14] [15]. Starting in 2014, they introduced *Virtualized FPGA Resources (VFRs)* to the OpenStack-based Canadian SAVI testbed. The FPGAs are accessed through a VM via PCIe passthrough and the partial reconfiguration of the VFRs is triggered through the Joint Test Action Group (JTAG) connection. As next step, the *"hardware abstraction layer"* and the *"hypervisor in the FPGA"* were extended to be compatible with the Xilinx SDAccel platform with OpenCL kernels on the host VM. In [16] the group demonstrates their framework with an acceleration of virtual network functions. The performance of the demonstrated system is limited by the throughput of the host VM.

The integration of an Ethernet controller into FPGAs is described by *Blott et al.* [17], who implemented a 10Gbps NIC and complete UDP and TCP offload engines. The authors achieved an increase of up to 36 times in requests per second per Watt in comparison to x86 servers with optimized software (SW). Also, their FPGA implementation of a key-value store resulted in a round-trip time between $3.5\mu s$ and $4.5\mu s$, which is an improvement of two orders of magnitude over standard x86 approaches.

In [18] *Knodel et al.* provided a detailed description of another approach for PCIe-attached FPGAs in the Cloud. They also present a comprehensive security model for such a kind of Reconfigurable Common Cloud Computing Environment.

In our previous work in [2][19][20][21] we advocated for the integration of a NIC into the FPGA logic (called iNIC), and for the deployment of such FPGAs as stand-alone resources via a direct attachment to the DC network. We refer to such connected FPGAs as stand-alone network-attached FPGAs or *disaggregated FPGAs*. We showed that iNICs enable denser packaging, lower power consumption and cost, and increased flexibility with respect to supported protocol stacks. We also demonstrated a distributed text analytics application with regular expressions and compared it to a SW-only implementation as well as to an FPGA-accelerated implementation with PCIe-attachment. The comparisons showed that the proposed network-attached FPGAs outperform the two other implementations by a factor of 40 regarding latency and a factor of 18 regarding the throughput [21].

III. System Architecture Proposal

Our goal is to deliver FPGA resources in the Cloud similarly to CPU VMs. This requires a cloud service that is easy-to-use, flexible and scalable.

To build our SW system part, we can learn from common cloud services — e.g. as defined in [22] — and build on TCP/IP as communication layer and the Representational State Transfer (REST) concept [23] as management and interaction principle, because both have proven to scale well. REST is a distributed system framework between nodes of arbitrary kind and designed to use a stateless communication protocol, typically the Hypertext Transfer Protocol (HTTP). The scalability of its stateless operation was demonstrated by CPU-based applications in the Web, and we anticipate the same for FPGAs, because the REST principles are platform independent. Furthermore, REST builds on uniform interfaces and enforces strong separations between the components. Web services that implement such an architecture are referred to as *RESTful*.

Additionally, in contrast to e.g. the cloud service of AWS [10, see "Step 3"], we want to allow the user to generate the FPGA bitstream under his/her exclusive control before uploading it to our cloud service. The goal is to protect the intellectual property of a user in the same way as a SW binary running on a VM.

This proposal builds on our prior work in [2] which describes the hardware (HW) of a dense FPGA platform that decouples the FPGA from the CPU of the server and connects the FPGAs directly to the DC network. This turns an FPGA into a disaggregated, stand-alone computing resource and renders the platform particularly cost and energy-efficient

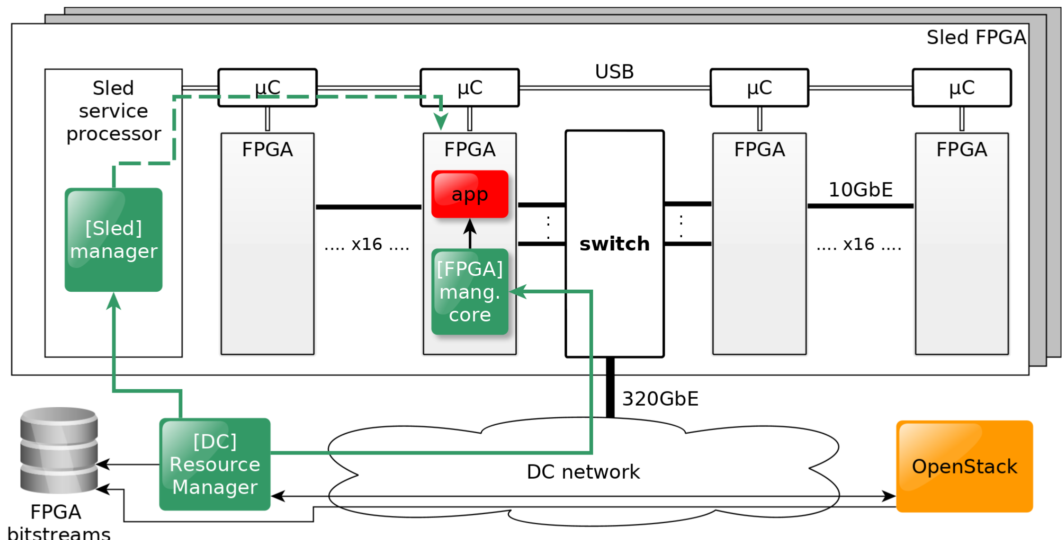

Fig. 1. System architecture for the Sled FPGA platform. 32 FPGAs, one switch and a service processor are combined on one carrier board and called *Sled* [2]. The management tasks are split into three levels — DC, Sled, and FPGA. A Sled is half of a 2U chassis. The OpenStack compute resources (Nova) as CPU nodes are also available for creating heterogeneous clusters.

because the number of deployed FPGAs becomes independent of the number of servers.

The platform in [2] integrates 32 FPGAs and a 64 port 10 Gb/s Ethernet switch with full cross-sectional bandwidth onto a passive water cooled carrier referred to as *Sled*. The switch of the Sled acts as a leaf switch that aggregates $32 * 10$ GbE links from the FPGAs and then connects to the core of the DC network via $8 * 40$ GbE up-links. The goal of this leaf-spin architecture is to avoid increasing the port density of the core switches. Each FPGA is mounted on a card that contains 16GB of DDR4 RAM, a Flash module to store the boot bitstream, and a tiny micro controller (μC) for powering the FPGA on/off as well as writing some configuration values to the FPGA. Additionally, each Sled contains a service processor to control the μCs and the Ethernet switch. The concept is illustrated in Fig. 1. We will refer to this platform as *Sled FPGA (SF)*.

With respect to the HW platform described above, we will now propose an infrastructure management framework for serving such network-attached FPGAs as virtual entities or clusters to users.

A. Requirements

Taking the above assumptions into account, we derived the following requirements for our system architecture:

R1) An application must only consist of a partial bitstream.
R2) An application must be independent of the physical FPGA, i.e. a hardware abstraction must be provided.
R3) The internal logic of the FPGA must be separated into privileged and non-privileged parts. The iNIC must be part of the privileged part.
R4) The privileged logic must always come from sources controlled by the provider.
R5) The FPGA card must be able to perform all operational tasks on its own, i.e. configure the application, set up the user network, administrate memory regions, and eventually set up the runtime environment.
R6) We assume a single physical communication channel between the FPGA and the DC network.
R7) All management tasks must be done with RESTful Application Program Interfaces (API).
R8) If possible, reuse existing DC services to execute (sub-) tasks.
R9) Multiple FPGAs must be able to connect with each other via TCP/IP (as well as with more than one CPU) to build a cluster.

B. High-Level Architecture

Our system architecture proposal is split into three levels (see Fig. 1): A *Data Center Resource Manager (DCRM)*, a *Sled Manager (SM)*, and an *FPGA Manager Core (FMC)*. There is one resource manager per DC to control many Sleds. The DCRM handles the user images and maintains a database of FPGA resources.

There is one sled manager for every 32 FPGAs. The SM runs on a service processor that is part of the Sled. It powers the FPGAs on and off, monitors the physical parameters of the FPGAs, and runs the SW management stack of the Ethernet switch.

There is one FMC per FPGA. The FMC contains a simplified HTTP server that provides support for the REST API calls issued by the DCRM.

In the end, the components of all levels work together to provide the requested FPGA resources in a fast and secure way.

For a cloud service based on SF, the necessary information about an *FPGA resource* at DC level is the FPGA type (e.g. XKU060), the IP address of the Sled service processor to which the FPGA belongs (see Fig. 1), the slot number

on that Sled (between 0 and 31), the type of available RAM on the module, and the current state of this FPGA (e.g. `AVAILABLE`, `USED`, or `MAINTENANCE`) . The DCRM maintains the database of FPGA resources containing such metadata. The resources can be added, modified and deleted by administrators. To allow a secure operation, the DCRM service also takes care of the authorization of users and the allowed usage quotas per user.

A user can upload *images*, i.e. FPGA bitstreams, which are also stored in the DCRM database. Each user can upload and delete her/his images.

If a user requests to configure one specific image to an FPGA, this image gets instantiated on an FPGA, i.e. configured, and then the combination of the two becomes an *instance*. Hence, the basic formula for our architecture is $\textit{FPGA resource} + \textit{image} = \textit{instance}$.

The DCRM supports heterogeneous *clusters* by assigning multiple FPGA instances and CPU VMs to the same subnetwork, providing unique node-ids for each instance, and propagating the necessary routing information to all devices as required by R9. The DCRM database also contains all the information about currently instantiated FPGA instances or clusters.

Operating under the control of the DCRM, the SM controls the power states of the FPGAs and writes the MAC and IP addresses into the FPGA, according to the API calls from the DCRM.

For SF, one single network interface carries both user and management traffic and the FPGA logic consequently needs to be separated into privileged and non-privileged (R3). Otherwise, the integrity of the DC network would be at risk, because the user could take control of the network interface to break out of her/his designated subnet, start denial-of-service attacks or harm the provider network in other ways.

The FMC controls the application and is part of the privileged logic. As mentioned before, we use PR to split the logic into privileged and unprivileged. To enable PR at runtime, it is necessary to define the placement and interface of the PR regions in advance [24]. Hence, PR adds further flexibility to the design, but also defines clear interfaces between the static and dynamic logic. Additionally, PR has the advantage of speeding up the deployment of partial bit files as well as protecting the user's intellectual property. The privileged logic is statically configured at power on and is referred to as the *Shell*. Therefore, by using PR we can control the privileged logic (R4) and define a hardware abstraction (R2). The Shell also provides runtime information, such as the node id if the FPGA is part of a cluster. The FMC is controlled by the DC resource manager via a RESTful HTTP API as defined by R7. Finally, the unprivileged application logic, also referred to as *Role*, is dynamically configured using a partial bitfile that is sent through the network every time a new PR is required.

Usually, to deploy an application on a compute cloud, the user need not take care of I/O-connections, the details of the hardware setup, or different versions of a carrier board. This should be true for FPGA clouds as well and is stated by R2. Consequently, the combination of Shell, Role, and the interface between them does not only separate the privileges within the FPGA but also abstracts the details of the network and memory hardware from the user as required by R2. This abstraction is provided by the Shell via industry-standard and easy-to-use AXI-Protocols [25].

IV. Implementation

In this section we describe our system architecture in detail and justify the design decisions based on the formulated requirements.

DCRM, SM, and FCM provide RESTful HTTP APIs. These APIs provide HTTP methods like `GET <entity-name>`, `POST <entity-name>`, or `PUT <entity-name>`. Usually, as defined in RFC 2616 [26], the `GET` method is for reading this entity, the `POST` method requests or adds a new entity, and the `PUT` modifies an existing entity.

A. DC-level management

New resources are introduced to the DCRM with a `POST /resource` request (R7). Only the DC provider is allowed to add or modify the resources and therefore the permission is checked first. For the user management the OpenStack *Keystone* module (see [27]) is used (R8). In our architecture, an FPGA is uniquely identified by the Sled-IP on which it resides and by the slot number on that Sled. Hence, the DCRM ensures that each such pair exists only once in the database before assigning a unique id as required by the REST-concept.

The user can upload his/her "FPGA image" using a `POST /image` method at the resource manager. The user must submit, besides the partial bitfile, the type of HW abstraction used — that is the type of Shell used (R2) — and the type of the FPGA board for which the bitfile was implemented. The OpenStack *Glance* module (see [27]) is used for storing the bitfiles and their metadata (R8).

Using a `POST /instance` or `POST /cluster` request (R7), the user informs the DCRM about the images he wants to deploy and which image should be assigned to which *node-id* in case of a cluster. Due to the use of a HW abstraction and PR, no other information is required and the images are mapped to the resources based on the metadata of the images. The DCRM then starts to power on the required FPGAs (if necessary), to distribute the network configurations using the SMs and to distribute the partial bitfiles of the application to the FMCs. Finally, the routing information within a cluster is propagated and the network setup is verified before the information about the topology is returned to the user. Again, instances and clusters are assigned a unique id after their submission and can be manipulated later based on this id.

The corresponding network abstraction of the FPGA instances, i.e. node-id and TCP/UDP stream, are also available as a SW library to be used on CPUs. As a consequence, application kernels on FPGAs and CPUs can easily talk to each other through using the node-ids.

B. Sled-level management

The concept of SF requires a service processor to manage the resources that are shared in every group of 32 FPGAs on a Sled. These include: two power converters, an Ethernet switch and a USB hub for communicating with the μCs located on every FPGA card.

This manager is controlled by the provider (R4) and it grants the Sled as a pool of 32 FPGAs to the DCRM. The SM controls the power on/off of the FPGAs and issues low-level management commands to the μCs (e.g. reset, write IP address, get temperature or program the Flash memory). The SM API is also designed in a RESTful manner (R7).

Due to security considerations, we decided to use the μCs to perform these tasks instead of using a combination of Wake-on-LAN (WoL) and Dynamic Host Configuration Protocol (DHCP). The "magic packets" for WoL are sent via layer 2, which can easily be exploited in a DC environment, unless we would introduce extra filters for layer 2 in the Sled. This solution appeared to entail more of an overhead to us and probably comes with additional implications compared to using small and cheap μCs for this task.

This μC should not be considered as a bus-attached processor. This device is always on and consumes less than 0.3 Watts. Its role is similar to a Baseboard Management Controller in providing basic functions such as power state, start-up, control and monitoring of the FPGA.

C. FPGA-level management

After the μC is done with assigning the FPGA its MAC and IP addresses, the remaining tasks to execute during deployment are performing the partial reconfiguration of the application, starting and resetting the application, and, if necessary, setting up the memory layout or routing information for the application. The FMC understands the following REST API calls issued by the DCRM:

- `POST /configure`: Submits a partial bitfile and triggers the PR of the *Role* region.
- `GET /status`: Returns some app-specific status information.
- `PUT /node_id`: Sets the node-id register of the Role.
- `POST /routing`: Sends the routing information of a cluster to the FPGA.

It checks the validity of the request and the CRC checksums within the partial bitfile and responds accordingly with the HTTP status code.

One core requirement of RESTful architectures is that one resource is managed by one single controller entity. This is necessary for an architecture in order to enable scaling and easy maintenance. Therefore, we introduced the FMC as third level of management to be responsible for all management modifications regarding the internal state of the FPGA, including the execution of the PR. This also enables the concurrent handling of FPGA resources, in contrast to using a serial bus like USB for 32 devices. As a result, with this RESTful management core inside the static design of the FPGA we fulfil the requirements 1, 5, and 7.

D. Shell-Role Architecture

Figure 2 gives an overview of our *Shell-Role-Architecture (SRA)* and the required functional cores inside an FPGA. The FPGA implements an Ethernet IP core from Xilinx to access the 10GbE network. The transport and network layers are implemented with TCP, UDP, Internet Control Message Protocol (ICMP) and IPv4 engines. The I/O component interfaces with the μC and subsequently with the SM. The connection to the DDR4 memory is also an IP core from Xilinx. Based on the network configuration, e.g. different VLANs or ports, the management traffic is forwarded to the FMC and the user traffic to the routing core (R6). The interface between the Role and the Shell is the abstract HW interface (R2). The `memory port` consists of two AXI4-master ports, the `node_id` is a 32-bit register, `start` and `reset` are one-bit signals. The `Network Interface` consists of several AXI4-stream connections for data and meta-data. The entire Shell is coded in HLS except for the above-mentioned Xilinx cores.

To deploy the Role, the FMC writes the received and verified PR bitfile to the Xilinx HWICAP core to trigger the PR (R1). While the PR is being executed, the FMC decouples all logic from the Role. The Role can be specified in VHDL, Verilog, HLS or any combination thereof.

Finally, when a Xilinx Debug Bridge IP Core [28] is inserted into a PR region, it can have its traffic forwarded via TCP for remote debugging in the Cloud. Our SRA comes along with a framework to compile and build user applications. A user only needs to issue one `make` command to get the application bitstream, without taking all the dependencies into account.

V. Evaluation

We evaluated our system architecture on the SF platform using Xilinx XCKU60 FPGAs.

Table I shows the overhead in terms of the FPGA resource usage for the runtime and the management cores. The resource usage for the FMC, which includes the REST implementation, and the Network Routing Core (NRC) are in the range of one to two percent of all available resources. This additional resource usage of the logic for the self-management of the FPGA is relatively low and therefore supports our stand-alone network-attachment proposal.

TABLE I
Resource usage of the FMC and NRC in a XCKU060

Resource	Available	Used		
		Total	FMC	NRC
LUT	331680	112891	1970	860
LUTRAM	146880	12795	55	16
FF	663360	140668	6729	2193
BRAM	1080	303	5	8

Table II compares the deployment time of a bitstream via a JTAG interface with that of our TCP-based approach. The JTAG speed for the experiments in Table II was set to $5\ \frac{Mbit}{s}$. As expected, the reconfiguration times of the partial bitfile are significantly shorter than the ones of the entire design.

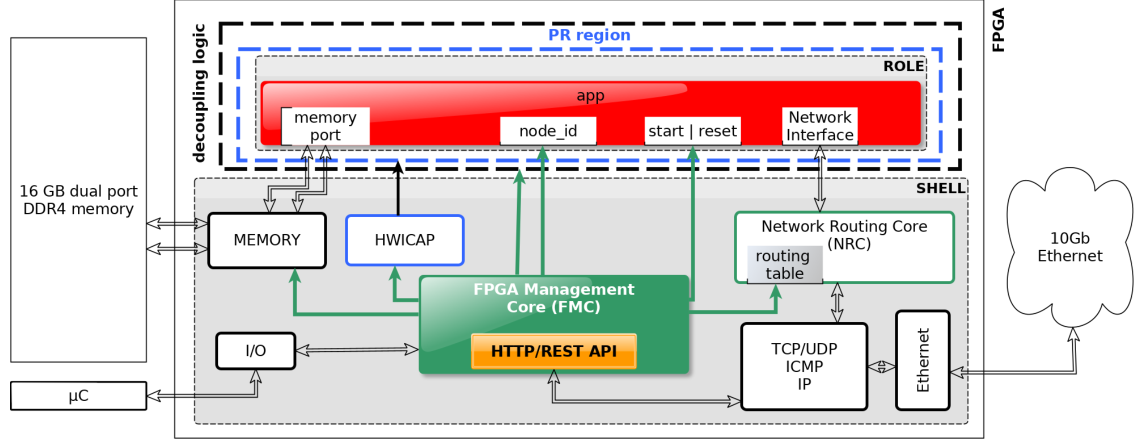

Fig. 2. Shell Role Architecture. All management tasks are controlled by the FPGA management core.

TABLE II
CONFIGURATION TIMES

operation	file size in MB	average time in *seconds*	average speed in $\frac{kB}{s}$
JTAG configuration of the complete design	24.1	42.8	560.747
`POST /configure` of partial bitfile over TCP	4.4	0.5	$8,264.112$
JTAG configuration of partial bitfile	4.4	8.2	535.714

Delivering the partial bitfile over TCP reduced this amount of time by a factor of 16.

Since the FPGA module manages itself and can run completely independently, SF in combination with the proposed system architecture is able to completely leverage the energy efficiency of FPGAs.

As a proof of concept, we set ourselves the goal of running a Message Protocol Interface (MPI) [29] application on a cluster of eight FPGAS and one CPU. In SW, the MPI program is compiled by the MPI compiler. Afterwards, in a cluster of $8+1$ CPUs, the command `mpirun` distributes the compiled application binary across the cluster, e.g. using Secure Shell (SSH) before executing it automatically [30][31].

To run the same program on our platform, we developed a cross-compiler that transforms parts of the MPI code into HLS-synthesizeable code, similar to the approach in [32]. This HLS code is subsequently imported as a Role into our SRA and the partial bitstream is generated automatically. Analogous to the SW-only process, our run-command uploads the resulting bitfiles and distributes them via the DCRM API. Next, the routing tables of the FPGAs are propagated and the network setup is verified. Listing 1 is the output of this run-command. This example demonstrates how an HPC application developer without specific FPGA experience is able to compile and execute an MPI application on a cluster of FPGAs.

VI. CONCLUSION

Agile and scalable management of large amounts of FPGAs is a key element for their further integration into data centers.

```
user@test0 $ ./SF_mpirun <path-to-SW-binary> <
path-to-PR-bitstream> 8

Upload PR bitfiles...
Creating FPGA cluster...
Id of new cluster: 1
Ping all nodes, build ARP table...
Starting MPI subprocess at 2019-03-21
21:26:30.455471

rank 1 addr: 10.12.200.17
rank 2 addr: 10.12.200.20
rank 3 addr: 10.12.200.19
rank 4 addr: 10.12.200.21
rank 5 addr: 10.12.200.22
rank 6 addr: 10.12.200.23
rank 7 addr: 10.12.200.24
rank 8 addr: 10.12.200.25
Here is rank 0, size is 9.
Distribute data and start client nodes.
[..........]
MPI execution time: 3.177521s
MPI subprocess finished at 2019-03-21
21:26:33.640393.
```

Listing 1. Snippet of one SF-MPIRUN execution example. In this case the user requested a cluster with eight FPGAs.

However, the flexibility and performance-per-cost advantages provided by FPGAs can only be leveraged in the Cloud if they can be deployed quickly and independent of other computing resources. In order to accommodate both, we proposed and implemented a system architecture for network-attached FPGAs in the Cloud using partial reconfiguration as one step towards making FPGAs true first-class citizens in the Cloud. The implemented end-to-end architecture realizes an easy-to-use cloud service, protects the intellectual property of users and enables heterogeneous clusters while complying with all derived requirements. Hence, it makes it possible to deploy an application on multiple FPGAs just by uploading a set of partial bitfiles and their corresponding clustering structure. To the best of our knowledge, this research is the first to implement a stand-alone PR over network and enables the RESTful management of network-attached FPGAs.

REFERENCES

[1] A. M. Caulfield *et al.*, "A cloud-scale acceleration architecture," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49, Taipei, Taiwan: IEEE Press, 2016, 7:1–7:13.

[2] F. Abel, J. Weerasinghe, C. Hagleitner, B. Weiss, and S. Paredes, "An FPGA platform for hyperscalers," *Proceedings - 2017 IEEE 25th Annual Symposium on High-Performance Interconnects, HOTI 2017*, pp. 29–32, 2017. DOI: 10.1109/HOTI.2017.13.

[3] L. Wang and W. Feng-yan, "Dynamic partial reconfiguration in FPGAs," *3rd International Symposium on Intelligent Information Technology Application, IITA 2009*, vol. 2, pp. 445–448, 2009. DOI: 10.1109/IITA.2009.334.

[4] A. Putnam *et al.*, "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services," *IEEE Micro*, vol. 35, no. 3, pp. 10–22, 2014. DOI: 10.1109/MM.2015.42.

[5] A. Linn. (2018). Real-time AI: Microsoft announces preview of Project Brainwave, [Online]. Available: https://blogs.microsoft.com/ai/build-2018-project-brainwave/.

[6] E. Chung *et al.*, "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018. DOI: 10.1109/MM.2018.022071131.

[7] Amazon.com, Inc. (2019). Amazon EC2 F1 Instances, [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/.

[8] IBM Research – China. (2014). OpenPOWER Cloud – Accelerating Cloud Computing, [Online]. Available: https://www.research.ibm.com/labs/china/supervessel.html.

[9] Amazon.com, Inc. (2019). Overview of AWS EC2 FPGA Development Kit, [Online]. Available: https://github.com/aws/aws-fpga.

[10] ——, (2019). AWS FPGA Hardware Development Kit (HDK), [Online]. Available: https://github.com/aws/aws-fpga/blob/master/hdk/README.md.

[11] Y. Lin and L. Shao, "SuperVessel: The Open Cloud Service for OpenPOWER," *White paper, IBM corporation*, 2015.

[12] F. Chen *et al.*, "Enabling FPGAs in the cloud," in *Proceedings of the 11th ACM Conference on Computing Frontiers - CF '14*, New York, New York, USA: ACM Press, 2014, pp. 1–10. DOI: 10.1145/2597917.2597929.

[13] S. Byma, J. G. Steffan, H. Bannazadeh, A. Leon-Garcia, and P. Chow, "FPGAs in the cloud: Booting virtualized hardware accelerators with OpenStack," *Proceedings - 2014 IEEE 22nd International Symposium on Field-Programmable Custom Computing Machines, FCCM 2014*, pp. 109–116, 2014. DOI: 10.1109/FCCM.2014.42.

[14] N. Tarafdar, T. Lin, N. Eskandari, D. Lion, A. Leon-Garcia, and P. Chow, "Heterogeneous virtualized network function framework for the data center," *2017 27th International Conference on Field Programmable Logic and Applications, FPL 2017*, 2017. DOI: 10.23919/FPL.2017.8056790.

[15] N. Tarafdar, T. Lin, E. Fukuda, H. Bannazadeh, A. Leon-Garcia, and P. Chow, "Enabling Flexible Network FPGA Clusters in a Heterogeneous Cloud Data Center," *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, pp. 237–246, 2017. DOI: 10.1145/3020078.3021742.

[16] N. Tarafdar, N. Eskandari, T. Lin, and P. Chow, "Designing for FPGAs in the Cloud," *IEEE Design and Test*, vol. 35, no. 1, pp. 23–29, 2018. DOI: 10.1109/MDAT.2017.2748393.

[17] M. Blott, K. Karras, L. Liu, K. Vissers, J. Bär, and Z. István, "Achieving 10Gbps line-rate key-value stores with FPGAs," *HotCloud (USENIX Workshop on Hot Topics in Cloud Computing)*, 2013.

[18] O. Knodel, P. R. Genssler, F. Erxleben, and R. G. Spallek, "FPGAs and the Cloud – An Endless Tale of Virtualization , Elasticity and Efficiency," *International Journal on Advances in Systems and Measurements,*, vol. 11, no. 3 & 4, 2018.

[19] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf, "Enabling FPGAs in hyperscale data centers," *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, pp. 1078–1086, 2016. DOI: 10.1109/UIC-ATC-ScalCom-CBDCom-IoP.2015.199.

[20] ——, "Disaggregated FPGAs: network performance comparison against bare-metal servers, virtual machines and linux containers," *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*, no. i, pp. 9–17, 2017. DOI: 10.1109/CloudCom.2016.0018.

[21] J. Weerasinghe, R. Polig, F. Abel, and C. Hagleitner, "Network-attached FPGAs for data center applications," *Proceedings of the 2016 International Conference on Field-Programmable Technology, FPT 2016*, pp. 36–43, 2017. DOI: 10.1109/FPT.2016.7929186.

[22] P. Mell and T. Grance, "The NIST Definition of Cloud Computing Recommendations of the National Institute of Standards and Technology," *National Institute of Standards and Technology, Information Technology Laboratory*, vol. 145, p. 7, 2011.

[23] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," PhD thesis, University of California, Irvine, 2000.

[24] Xilinx Inc., "Vivado Design Suite User Guide: Partial Reconfiguration (UG909) [2017.4]," Tech. Rep., 2017.

[25] ARM, "AMBA AXI and ACE Protocol Specification," *Arm*, pp. 1–306, 2011.

[26] Fielding et al. (1999). RFC2616 – Hypertext Transfer Protocol – HTTP/1.1, [Online]. Available: https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html.

[27] OpenStack Community. (2019). OpenStack Component Map, [Online]. Available: https://www.openstack.org/assets/software/projectmap/openstack-map.pdf.

[28] Xilinx Inc., "Debug Bridge v3.0 LogiCORE IP Product Guide PG245," Tech. Rep., 2018.

[29] D. W. Walker and O. Ridge, "MPI : A STANDARD MESSAGE PASSING INTERFACE," *CONTRIBUTION TO TOP 500 REPORT*, 1995.

[30] OpenMPI Community. (2019). Open MPI Documentation, [Online]. Available: https://www.open-mpi.org/doc/.

[31] MPICH Community. (2019). MPICH Documentation, [Online]. Available: http://www.mpich.org/documentation/guides/.

[32] N. Eskandari, N. Tarafdar, D. Ly-Ma, and P. Chow, "A modular heterogeneous stack for deploying fpgas and cpus in the data center," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19, Seaside, CA, USA: ACM, 2019, pp. 262–271. DOI: 10.1145/3289602.3293909.