The Feniks FPGA Operating System for Cloud Computing

Jiansong Zhang[§] Yongqiang Xiong[§] Ningyi Xu[§] Ran Shu[§†] Bojie Li[§‡]
Peng Cheng[§] Guo Chen[§] Thomas Moscibroda[§]

[§] Microsoft Research [†]Tsinghua University [‡]USTC

{jiazhang,yqx,ningyixu,v-ranshu,v-bojli,pengc,guoche,moscitho}@microsoft.com

ABSTRACT

Driven by explosive demand on computing power and slowdown of Moore's law, cloud providers have started to deploy FPGAs into datacenters for workload offloading and acceleration. In this paper, we propose an operating system for FPGA, called Feniks, to facilitate large scale FPGA deployment in datacenters. XFeniks provides abstracted interface for FPGA accelerators, so that FPGA developers can get rid of underlying hardware details. In addtion, Feniks also provides (1) development and runtime environment for accelerators to share an FPGA chip in efficient way; (2) direct access to server's resource like storage and coprocessor over PCIe bus; (3) an FPGA resource allocation framework throughout a datacenter. We implemented an initial prototype of Feniks on Catapult Shell and Altera Stratix V FPGA. Our experiements show that device-to-device communication over PCIe is feasible and efficient. A case study shows multiple accelerators can share an FPGA chip independently and efficiently.

1 INTRODUCTION

Driven by explosive demand on computing power and slowdown of Moore's law, heterogeneous computing has attracted huge interests recent years. In the case of cloud computing, service providers are eager to offload large amount of CPU loads to more power/cost efficient devices, such as GPU, FPGA and ASIC, so as to support emerging workloads like deep learning inference and training, as well as save cost for existing workloads. Among these computing devices, FPGA can provide the highest flexibility in addition to much higher power/cost efficiency than CPU. Thus, many cloud providers have decided to deploy FPGA in large scale. For example, Microsoft has started to deploy FPGA in every Azure server to accelerate Bing ranking [4], network virtualization [6], and other workloads; Amazon has started to provide special EC2 instances mounting multiple FPGAs to cloud users [2]; Baidu has deployed FPGAs to accelerate SSD access in its cloud [18].

FPGA contains a large number of basic logic units, *e.g.*, LUT, flip-flop, block memory and DSP, as well as rich interconnections between the units. In theory, an FPGA chip can be configured to any type of hardware logic, even processors like CPU, GPU, network processor, *etc.* In practice, a set of workloads which can take advantage of FPGA's high parallelism and flexible data-width will most likely be offloaded, such as ranking [20], compression [8], encryption [4, 13], pattern matching [21], deep learning serving [17], *etc.*

From cloud providers' point of view, they expect those deployed FPGAs to accelerate as many cloud workloads as possible to pay back the investment, meanwhile also catch up the pace of workload evolving. Therefore, cloud providers desire various infrastructure supports to facilitate workload offloading. Firstly, to achieve high productivity, FPGA developers should be able to focus on application logic, but get rid of underlying details of specific hardware, like off-chip memory controller, PCIe endpoint and DMA engine, network protocols. Secondly, as coprocessors, FPGAs should be able to access cloud resources in an easy and efficient way. Cloud resources include server's main memory, disk or SSD storage, other coprocessors like GPU and many-core processor (e.g., Intel Xeon Phi), and cloud networks. Thirdly, as a new type of cloud resource, FPGA itself also should be allocated, scheduled and accessed in an easy and efficient manner.

In this paper, we present our research effort towards such infrastructure support and propose an operating system layer for FPGA, called Feniks. Basically, Feniks provides abstracted interfaces to various FPGA accelerators. On Feniks, accelerator developers can focus on accelerator logic itself but get rid of underlying details of FPGA IOs like FPGA to host, FPGA to off-chip memory, FPGA to storage and FPGA to network interface card communications. Moreover, Feniks separates operating system and application accelerators by leveraging the partial reconfiguration feature provided by FPGA vendors, so that OS and application images can be loaded separately. This separation makes it possible for cloud providers to take full control of the FPGA hardware and physical interfaces, and perform protection for malicious or careless accelerators from destroying other FPGA logic or host system.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

APSys '17, Mumbai, India
© 2017 ACM. 978-1-4503-5197-3/17/09…$15.00
DOI: 10.1145/3124680.3124743

In additon to hardware abstraction and OS/application separation, Feniks provides three important features. Firstly, Feniks can further divide an FPGA chip into multiple independent regions. In this way, multiple accelerators will share the same FPGA chip without interfering with each other. Feniks also provides IO virtualization so that multiple accelerators can use identical virtual IO interface and get similar IO performance. Secondly, Feniks provides direct access to server's resources like disk and other coprocessors over server's PCIe bus. In this way, FPGA can communicate to devices without CPU intervention, thereby saving CPU cycles and reducing communication latency when FPGA accelerator needs to write data into disk or work together with other coprocessor to construct a computing pipeline. Moreover, by connecting cloud network directly, FPGA can also access resources in remote servers. Thirdly, Feniks provides a resource allocation framework for FPGAs throughout a datacenter. Applications can use this framework to obtain available FPGA resources and deploy accelerator for workload offloading.

We implemented an initial prototype based on Catapult Shell and Altera Stratix V FPGA. The operating system components occupies 13% logic and 11% on-chip memory. Our experiments using two FPGAs prove that PCIe root complex can provide near full PCIe capacity for device-to-device communication and sub-*us* latency. A case study with data compressor and network firewall shows that multiple accelerators can share an FPGA chip without interfering with each other. Finally, accelerator migration based on Feniks's resource sharing framework takes less than 1s between two servers on the same rack.

2 BACKGROUND AND RELATED WORK

Deploying FPGAs in cloud servers is becoming a trend [2, 4, 18]. Normally, each FPGA is carried on a board with one or more DRAM modules attached. The board is then inserted into a server's PCIe slot. The FPGA can communicate with server's CPU through interrupt and shared memory, *i.e.*, both in server's main memory and in FPGA's on-chip or off-chip memory which is mapped in server's address space. Depending on specific deployment strategy, the FPGA board may also contain one or more network interfaces connected to cloud network [4, 20] or certain dedicated wires [2, 20].

Implementing these interfaces and necessary upper layer logic, *e.g.*, direct memory access over PCIe endpoint, network transport over Ethernet MAC, *etc.*, also consumes FPGA's common logic units, and requires developers' effort to build up. Fortunately, FPGA boards usually share the same configuration across a cloud to ease large scale deployment, it is possible to pack FPGA's interface logic into a fixed framework, which is usually called FPGA shell, *e.g.*, in Microsoft Catapult [20] and in Amazon EC2 [2]. In academia, there is also effort like RIFFA [11] which provides a framework for similar purpose but aims higher to adapt to diverse hardware configurations. In this paper, we extend the shell concept to operating system concept by adding a set of advanced features like performance isolation between applications and operating system, efficient cloud resource access and flexible FPGA resource allocation. LEAP [7] also brings operating system concept but extends in a different way by providing programming model and compiler to automatically generate FPGA design from application modules and supporting libraries. This effort is more aligned with high level programming support provided by Xilinx and Altera, as well as other academia efforts like Bluespec [16], Hthreads [19], ClickNP [13], CMOST [26], *etc.*

FPGA resource sharing and allocation in cloud has started to attract research interests. On the one hand, Byma [3] and Chen [5] share a single FPGA chip to multiple users by dividing logic units in an FPGA chip into several virtual accelerators using partial reconfiguration, and then allocating virtual accelerators to users using openstack. On the other hand, FPGAs are grouped together to construct larger accelerator. For example, Catapult [20] connects every 48 FPGAs into a cluster using a secondary cross-bar network. Amazon [2] connects 8 FPGAs in a ring topology using dedicated wires. In academia, FPGA cluster generator [23] is proposed to group FPGAs over network by leveraging SAVI, openstack and Xilinx SDAccel. In Feniks, we provide a framework for flexible FPGA resource allocation throughout a datacenter. This framework allows multiple applications to share the same FPGA chip, as well as grouping multiple FPGAs to serve a single application under certain latency and bandwidth constraints.

Finally, there is a rich set of literatures which integrate FPGA into general purpose operating system. For example, BORPH [22] modified Linux kernel to run FPGA process in the same way of running CPU process. HybridOS [12] also modified Linux to provide a framework for CPU and FPGA accelerator integration. ReconOS [14] extends multi-thread programming model into hybrid CPU and FPGA platform. FUSE [10] leverages loadable kernal module to support FPGA logic changes while integrating with software operating system. We notice that, most of the integration works above are implemented in embedded platforms that FPGA is close to CPU. In our design, we do not incorporate tight integration between software and FPGA operating system. Because in today's cloud deployment, FPGAs reside in servers' IO domain and suffer from larger latency while communicating with CPU, *i.e.*, it will be inefficient if FPGA and CPU communicate as frequently as multiple CPUs or multiple cores. Nevertheless, we expect in the future, when FPGA is integrated into CPU socket [9] or even CPU die, the integration between software and FPGA operating system will become more desirable and critical.

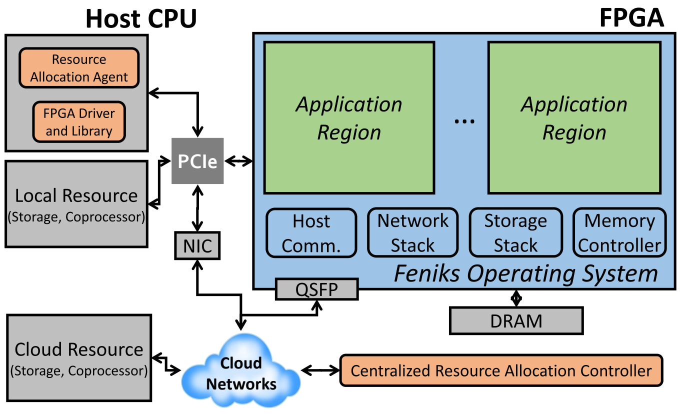

Figure 1: Feniks operating system overview. Feniks provides abstracted interfaces to applications by dividing an FPGA into an OS region and several application regions. The OS region contains stacks and modules to communicate with FPGA's local DRAM, host CPU and memory, server resources and cloud resources in an efficient manner. Feniks also includes support for FPGA resource allocation with centralized controllers in cloud and agents running on host CPUs.

3 FENIKS FPGA OPERATING SYSTEM

In this section, we present the design of Feniks FPGA operating system which provides infrastructure support to facilitate the development and opertaion of FPGA accelerators in cloud. As shown in Figure 1, on each FPGA chip, a Feniks instance divides an FPGA's space into an OS region and one or several application regions. Feniks provides FIFO based interfaces for each application region to use off-chip DRAM, communicate with host application instance and access various cloud resources. Accelerator developers only need to connect their accelerator logic to these abstracted interfaces without worrying about the detailed implementation of underlying hardware interfaces, therefore can focus on developing accelerator logic. On runtime, OS instance is loaded separated with accelerator instances. Normally, OS instance is loaded in advance by cloud operator and rarely changed. Then, accelerators can be loaded dynamically by users. In section 3.1 we will further discuss performance isolation between accelerators.

Besides basic OS functions, a key design goal of Feniks is to facilitate resource access and allocation for FPGAs in cloud. On the one hand, Feniks fully exploits the connectivity over server's PCIe bus to enable FPGA to directly access devices attached in server, such as storage device and coprocessors. In section 3.2 we will further discuss the techniques for cloud resource access over PCIe. On the other hand, Feniks also provides support for FPGA resource allocation to cloud users and applications. Specifically, Feniks always launches a resource allocation agent on host CPU to allocate and load accelerators. These agents execute commands from centralized controllers which perform global FPGA resource allocation and scheduling for a datacenter. In section 3.3 we will elaborate FPGA resource allocation in Feniks.

3.1 Performance Isolation and Multi-tasking

Although Feniks resembles software operating system in function, its implementation is necessarily very different as it targets FPGA. In software operating system running on CPU, user programs are organized into processes and threads that share a common execution substrate with operating system, *i.e.*, the processor and its memory. FPGAs differ from this model in the way that FPGA executions are multiplexed in spatial domain instead of time domain, *i.e.*, FPGA programs are organized as spatially distributed modules, with portions of the FPGA fabric dedicated to each of the different functions of the program and the operating system. Therefore, performance isolation in Feniks is natually performed by isolating application regions and OS regions. Similarly, multi-tasking is supported by assigning tasks into multiple application regions in which tasks can run simultaneously.

We leverage the *partial reconfiguration* (PR) feature provided by FPGA vendors. PR basically disables the logic units and interconnects on the boundary of a specified region, therefore physically prevents logic inside and outside the region from interfering with each other. In order to connect a PR region with outside logic, some LUTs can be explicitly enabled in the boundary specification. In Feniks, we provide a set of templates to accelerator developers with different PR region configurations. For example, single PR region for application to occupy an FPGA exclusively, or multiple PR regions for applications to share an FPGA. Accelerator developer only need to select a proper template and fill in the accelerator logic. After compilation, an image containing only accelerator logic will be generated. To deploy the image, a Feniks image with the same PR region configuration should be loaded in advance, and then accelerator image can be loaded any time later. In this way, cloud operators are possible to provide security support that the operating system functions will not be destroyed by malicious or careless accelerator logic, and multiple accelerators will not interfere with each other.

Feniks mainly relies on spatial sharing for multi-tasking instead of dynamic accelerator reloading (context switching) because application image loading time will add significant overhead. As shown in Figure 2, application loading time measured on Altera Stratix V FPGAs is between 10*s ms* and 100*s ms* which is proportional to the application region size. It is reasonable because loading an application needs to reconfigure all the logic units in the region. However, we expect context switching in the same region would be feasible if multi-context FPGA [24] is deployed sometime.

Finally, in Feniks, we leverage the ability of dynamic accelerator loading to provide application migration service.

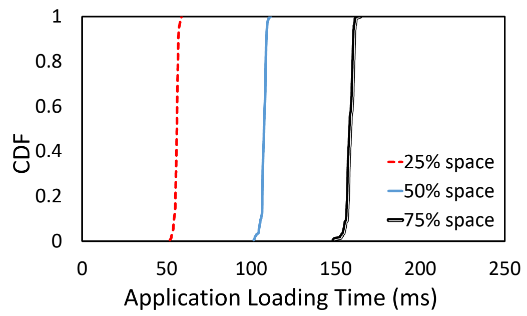

Figure 2: Accelerator loading time to PR region when region size is 25%, 50% and 75% of FPGA space. The loading time is between $10ms \sim 100ms$ and proportional to region size. Due to the high loading time, we do not encourage multi-tasking using context switching.

Specifically, when migration decision is made (as will be discussed in section 3.3), the running accelerator stores its states into on board memory. Then, both the stored states and accelerator image are transmitted to destination host, on which the states and image are loaded into destination FPGA's application region.

3.2 Accessing Server and Cloud Resources

In this subsection, we discuss in details the operating system modules in Feniks. As a key design goal, we emphasize how Feniks enables FPGAs to access server and cloud resources in an efficient way, including direct access to local resources over PCIe and remote access through cloud networks.

3.2.1 Local Direct Access over PCIe. In today's cloud, FPGAs acts as coprocessors in cloud servers. By default, FPGA does not have direct access to various resources in server's IO domain like disk and other coprocessors. BORPH [22] enables FPGA to access Linux files by adding kernel service to receive FPGA's commands and execute instead. However, this approach is inefficient as CPU will be heavily involved. For example, when we offload data compression engine into FPGA and would like to write the compressed data into disk, it will add significant CPU overhead if FPGA first writes compressed data into main memory through DMA and then CPU writes the data into disk. Similarly, if we want to build a computing pipeline using FPGA and GPU in the same server, *e.g.*, using FPGA to decompress a big data set before performing deep learning model training using GPU, it will also add significant CPU overhead as well as more latency if every FPGA and GPU need to write intermediate results into main memory and ask CPU to forward.

In Feniks, we leverage the device-to-device connectivity over server's PCIe bus to enable efficient resource access. As shown in Figure 3, various devices are connected to CPU through PCIe interface. Every device implements a PCIe endpoint which can communicate with the PCIe root complex

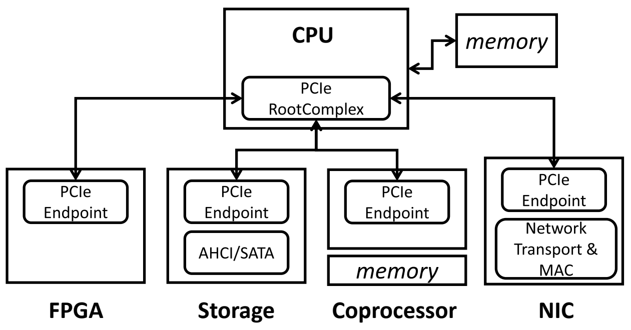

Figure 3: Devices are connected to CPU's PCIe root complex. Traditionally, devices send data to main memory through DMA, and CPU will forward to other devices. However, PCIe root complex actually supports device-to-device communication. Every device can send messages to others through their memory mapped IO address.

inside CPU. Since every device has a PCIe configuration space mapped in software operating system's memory address space, one device can also directly communicate with other devices using their memory-mapped PCIe configuration space address. This connectivity has been exploited in GPU to RDMA NIC direct connection [1].

Therefore, in Feniks, we add modules in FPGA operating system to enable accelerators to access various devices directly through PCIe. Among all the devices, the easiest is FPGA to FPGA communication when multiple FPGAs are inserted in the same server. Every FPGA only needs to get other FPGA's configuration space address and uses DMA-write message defined in PCIe transport to send data. This FPGA-to-FPGA communication has also been provided by Amazon's single server FPGA cluster [2]. Accessing coprocessors are also relatively easy as they usually map their memory into their PCIe configuration space, therefore FPGA can directly write data into coprocessors' memory using DMA-write messages. Meanwhile, in opposing direction, coprocessors can also use their own DMA engine to write data into FPGA's PCIe configuration space. Storage devices also can be access through PCIe. Specifically, When FPGA gets AHCI's PCIe configuration space address, it can send read and write messages to AHCI's registers to send commonds. In this way, FPGA can read and write any sector of the attached storage devices. However, to avoid racing between software OS and FPGA OS, in Feniks, we currently reserve a portion of disk space dedicated for FPGA to access. To enable accelerators to use this reserved disk space, in our design we include a simplified file system similar to [15] for accelerators to create, read and write files. Network interface card (NIC) is also attached in PCIe slots and therefore can be used by FPGA. Traditional NIC requires complicated IP and transport layer network stack implementation in software operating system. The stack contains complicated control logic like TCP,

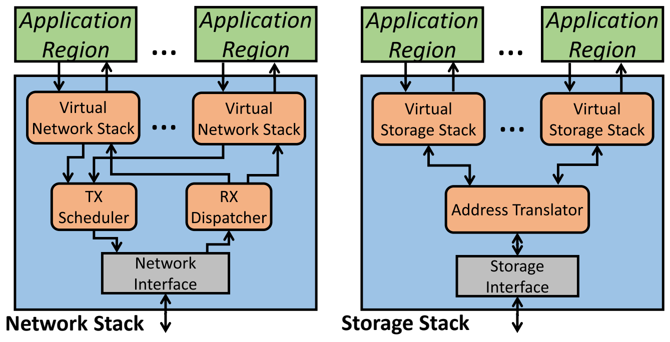

Figure 4: Feniks provides virtualized devices and stacks to application regions. In this way, every accelerator can use identical device interface and address space.

therefore is very difficult to implement in FPGA. Fortunately, recent advance of hardware based stack implementation in RDMA NIC greatly simplifies the NIC interface and makes it possible for FPGA to use. Similar to GPUDirect [1], FPGA needs to send its own PCIe configuration space address to RDMA NIC to perform remote DMA read and write for networking with other servers. Worth noting, all the direct device access requires driver support in software operating system as they need to exchange memory mapped IO address and reserve resource to avoid racing.

3.2.2 Remote Access Through Cloud Networks. As long as FPGAs can connect with each other through cloud networks, each FPGA can act as an agent for remote FPGA to access its local server's resources. For example, when multiple FPGAs across servers are grouped together to construct a computing acceleration pipeline, *e.g.*, for search ranking [20], or an FPGA needs to read data from a remote disk.

Besides RDMA NIC, the network connectivity also can be achieved through the network interface available on FPGA chip itself. For example, Microsoft Catapult provides such a topology called "bump-in-the-wire" in which FPGAs are connected directly with each other through cloud networks [4]. In such design, a RDMA like hardware transport should be implemeted in FPGA to control packet transmissions.

3.2.3 FPGA IO Virtualization. Finally, for all the FPGA interfaces, *e.g.*, network, storage, host communication and off-chip memory, multiple application regions must use the same underlying device and stack for I/O operations. To support I/O resource sharing and provide identical interface to all application regions, Feniks incorporates device and stack virtualization. Figure 4 shows the structures for network and storage virtualization. Both stacks provide virtual stack instances separately for every application region. These virtual instances are connected to underlying device through multiplexing logics. In network stack, transmissions are divided into two directions, *i.e.*, TX and RX. On TX direction, since aggregated input bandwidth will exceed output bandwidth, we should provide mechanism for certain quality of service. In Feniks, we schedule traffics in TX scheduler according to certain network sharing policy, *e.g.*, weighted fair bandwidth sharing. On RX direction, a dispatcher is enough which dispatches incoming network packets to corresponding virtual stack instance. In storage stack, we rely on address translator to perform storage resource sharing. For block device, *i.e.*, disk or SSD, we provide identical virtual sector address space to every application region, and then use address translator to translate virtual sector address into physical sector address.

For off-chip memory, in order to save more logic resource for application regions, we do not include caching structure in Feniks's operating system region as have been done in other design [7]. But we leave the raw memory interface with only necessary address translation for multiple accelerators to use identical virtual memory space.

For host communication interface, we leave a DMA interface and a register interface for every application region. On the DMA interface, host memory address information is not passed but left in FPGA operating system, so that accelerators will not perform DMA to illegitimate address which may destroy the software operating system. On the register interface, the register address space is also identical to every application region and the underlying operating system will perform address translation.

3.3 Support for FPGA Resource Allocation

In this subsection, we discuss the resource allocation framework in Feniks. As discussed in Section 3.1, the basic unit of FPGA resource is application region. Depending on specific requirement, an application can occupy one or multiple regions, and the region size can be selected from a set of configurations. Feniks's resource allocator will load different operating system image for different region size, as also has been discussed in Section 3.1.

Feniks manages FPGAs in a manner similar to Yarn [25] and other job schedulers. As shown in Figure 5, a logically centralized resource allocation controller tracks FPGA resources throughout the cloud. For each specific application, a service manager will request FPGA resources from central controller through a lease-based model. Then the service manager sends configuration commands to the resource allocation agents reside on every server node. According to the commands, these agents will load proper OS image and set up inter-FPGA connections. On application serving period, the agents also load accelerator images dynamically and monitor system status continuously.

In many cases, an application only requires a single FPGA region to accelerate the workload on host server, for example, data compression [8], network virtualization [6], pattern matching [21], *etc.* Central controller will prefer to allocate

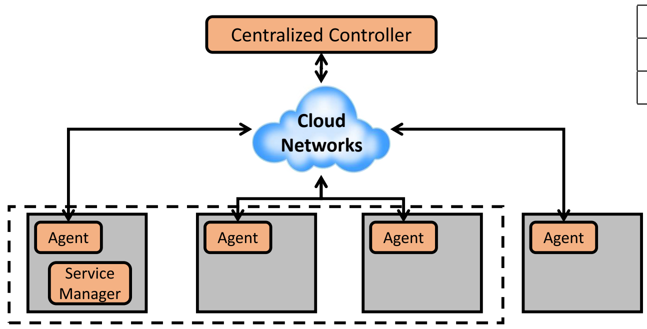

Figure 5: Feniks's resource allocation framework. A logically centralized controller tracks FPGA resources throughout the datacenter. On every server, an agent loads proper FPGA images and set up inter-FPGA connections according to the commands from service manager. A service manager may group multiple FPGAs into a pipeline depending on application requirement.

regions from the FPGAs which are already serving other applications. In these cases, service manager needs to specify IO bandwidth requirement to guide the configuration of the schedulers in FPGA operating system, as described in Section 3.2.3. In other cases that an application instance requires more than one FPGA to serve, its service manager needs to specify latency and bandwidth requirements for grouping FPGAs. For example, for latency sensitive application like search ranking [20] and deep learning inference [17], using those FPGAs in the same server or rack, or interconnected with additonal dedicated wires [2, 20], will be prefered.

4 PRELIMINARY RESULTS

We implemented an initial prototype of Feniks based on Catapult Shell and Altera Stratix V FPGA. Our prototype includes a streaming DMA engine, network (including FPGA-to-FPGA and FPGA-to-coprocessor connection) stack, off-chip memory controller, IO virtualization modules (as described in section 3.2.3) and partial reconfiguration engine. Table 1 shows resource consumption numbers of these operating system components. In total, our current implementation of Feniks's operating system consumes 13% logic and 11% on-chip memory of the Stratix V FPGA. Although not implemented yet, we expect that the storage stack and the rest of network stack will add limited overhead because they are request-response interfaces similar to DMA engine and not more complicated. Moreover, using later catapult hardware with Arria 10 FPGA which contains 2.5 times more logic and BRAM, Feniks's operating system will occupy less portion.

4.1 Communication over PCIe

Based on our prototype, we tested the communicate capability over PCIe. We inserted two FPGA boards into Dell R730

	DMA&Network	DDR Controller	PR engine
Logic (ALM)	4.8%	7.6%	0.2%
Block RAM	7.1%	3.5%	0.4%

Table 1: Resource consumption of operating system components in our initial prototype.

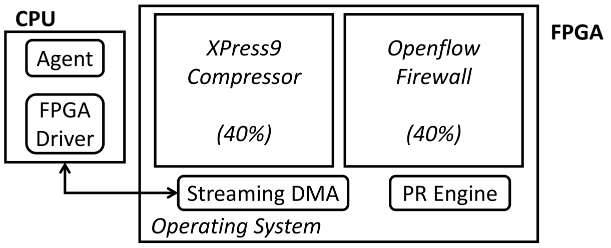

Figure 6: Case study: Feniks supports data compressor and network firewall to run simultaneously and independently. Application migration is decided by central controller on CPU and executed by agent on FPGA.

server (two Intel Xeon E5-2698 CPUs)'s PCIe slots. We test communication throughput and round-trip latency when two boards are attached to the same CPU or different CPUs. We found the PCIe root complex provides nearly full capacity (3.9GBps, PCIe gen3 x8) for devices to communicate over PCIe, but the QPI interconnection between CPUs will be the throughput bottleneck (0.25GBps) for device-to-device communication, though round trip latency is always as low as around 1us in both cases.

We conclude that device-to-device communication over PCIe is feasible and beneficial by avoiding CPU overhead and reducing latency. To optimize performance, we suggest to attach devices to the same CPU or use PCIe switch chip as the same observation from GPUDirect [1].

4.2 Case Study

Then we discuss an example use case in which two accelerators, *i.e.*, a data compression engine and a network firewall, are sharing the same FPGA chip on top of Feniks.

For applications, we use the XPress9 compressor [8] implemented in verilog and openflow firewall [13] implemented in opencl. As shown in Figure 6, in this case we allocate 40% of FPGA space to each of the applications which is already sufficient. We customized both applications from their original implementations to fit in the 40% regions. The customized XPress9 compressor provides lossless data compression and achieves 6% better compression ratio and 10x more throughput than software based GZip compression with level 9 (the best) optimization on a single Intel Xeon CPU core. The customized openflow firewall provides 20x more throughput than Linux IPTables and 3x more throughput than Click+DPDK implementation which are both on Intel Xeon CPU. These two applications are both throughput heavy, but aggregately they have not exceeded DMA bandwidth. The peak load of compressor and firewall are 10.6Gbps and 19.8Gbps, respectively, while the underlying DMA bandwidth is 48Gbps on single PCIe endpoint and 96Gbps on two PCIe endpoints. Therefore, the scheduler (Section 3.2.3) in DMA virtualization is reduced to round-robin sheduling.

We also tested application migration using resource allocation framework. The process is performed as follows. The service manager of the application (section 3.3) first makes migration decision.It then notifies agents on both source FPGA and destination FPGA. The source agent notifies the running accelerator to store its states into off-chip memory. Then the source agent turns the accelerator off and transmits stored states to destination FPGA. On the mean time, destination agent loads accelerator image. Upon receiving the states from source agent, destination agent turns the accelerator on and the migration is completed. In our test, the migration time is less than 1s for both above applications when source and destination server are in the same rack, in which the image loading time is around 70ms.

5 CONCLUSION

In this paper, we present the design of Feniks, an FPGA operating system which provides infrastructure support to facilitate cloud workload offloading. In addition to abstracted interface, Feniks provides (1) development and runtime environment for multiple accelerators to share an FPGA chip in an efficient way; (2) direct access to server's resource over PCIe bus; (3) an FPGA resource allocation framework throughout a datacenter. As a research project keeping improved, we believe the development of Feniks will benefit the use of FPGA in cloud computing.

ACKNOWLEDGEMENT

We would like to thank Kun Tan, Larry Luo, Derek Chiou, Andrew Putnam and Tong He for their initial exploration which provides valuable experience for the system design. We also would like to thank the anonymous reviewers for their insightful and constructive comments.

REFERENCES

[1] 2016. NVIDIA GPUDirect. *https://developer.nvidia.com/gpudirect* (2016).

[2] 2017. AWS EC2 FPGA Hardware and Software Development Kit. (2017). https://github.com/aws/aws-fpga

[3] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow. 2014. FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack. 109–116.

[4] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. 2016. A cloud-scale acceleration architecture. In *MICRO 2016*. IEEE, 1–13.

[5] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. 2014. Enabling FPGAs in the Cloud. In *CF 2014*. ACM, New York, NY, USA, Article 3, 10 pages.

[6] Daniel Firestone. 2017. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud.. In *NSDI*. 315–328.

[7] Kermin Fleming, Hsin-Jung Yang, Michael Adler, and Joel Emer. 2014. The LEAP FPGA operating system. In *FPL 2014*. IEEE.

[8] Jeremy Fowers, Joo-Young Kim, Doug Burger, and Scott Hauck. 2015. A scalable high-bandwidth architecture for lossless compression on fpgas. In *FCCM 2015*. IEEE, 52–59.

[9] Prabhat K Gupta. 2015. Xeon+ fpga platform for the data center. In *ICARL 2015*.

[10] A. Ismail and L. Shannon. 2011. FUSE: Front-End User Framework for O/S Abstraction of Hardware Accelerators. In *FCCM 2011*.

[11] Matthew Jacobsen, Dustin Richmond, Matthew Hogains, and Ryan Kastner. 2015. RIFFA 2.1: A Reusable Integration Framework for FPGA Accelerators. *ACM Trans. Reconfigurable Technol. Syst.* (2015).

[12] John H Kelm and Steven S Lumetta. 2008. HybridOS: runtime support for reconfigurable accelerators. In *FPGA 2008*. ACM.

[13] Bojie Li, Kun Tan, Layong Larry Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, and Peng Cheng. 2016. Clicknp: Highly flexible and high-performance network processing with reconfigurable hardware. In *SIGCOMM 2016*. ACM.

[14] E. Lubbers and M. Platzner. 2007. ReconOS: An RTOS Supporting Hard-and Software Threads. In *FPL 2007*.

[15] Ashwin A. Mendon, Andrew G. Schmidt, and Ron Sass. 2009. A Hardware Filesystem Implementation with Multidisk Support. *Int. J. Reconfig. Comput.* 2009 (Jan. 2009).

[16] Rishiyur S. Nikhil. 2008. *Bluespec: A General-Purpose Approach to High-Level Synthesis Based on Parallel Atomic Transactions*. Springer Netherlands, Dordrecht, 129–146.

[17] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, et al. Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?. In *FPGA 2017*.

[18] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. 2014. SDF: Software-defined Flash for Web-scale Internet Storage Systems. In *ASPLOS 2014*. ACM.

[19] Wesley Peck, Erik Anderson, Jason Agron, Jim Stevens, Fabrice Baijot, and David Andrews. 2006. Hthreads: A computational model for reconfigurable devices. In *FPL 2006*. IEEE.

[20] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA 2014*.

[21] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. 2017. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In *ICMD 2017*. ACM.

[22] Hayden Kwok-Hay So and Robert W Brodersen. 2006. Improving usability of FPGA-based reconfigurable computers through operating system support. In *FPL 2006*. IEEE.

[23] Naif Tarafdar, Thomas Lin, Eric Fukuda, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. 2017. Enabling Flexible Network FPGA Clusters in a Heterogeneous Cloud Data Center. In *FPGA 2017*. ACM.

[24] Steven Trimberger, Dean Carberry, Anders Johnson, and Jennifer Wong. 1997. A time-multiplexed FPGA. In *FCCM 1997*. IEEE.

[25] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache hadoop yarn: Yet another resource negotiator. In *SoCC 2013*. ACM.

[26] Peng Zhang, Muhuan Huang, Bingjun Xiao, Hui Huang, and Jason Cong. 2015. CMOST: A System-level FPGA Compilation Framework. In *DAC 2015*. ACM.