Using FPGAs as Microservices: Technology, Challenges and Case Study

David Ojika[1 2], Ann Gordon-Ross[1], Herman Lam[1], Bhavesh Patel[2], Gaurav Kaul[3], Jayson Strayer[3]

[1]University of Florida [2]DELL EMC [3]Intel Corporation

david_ojika, bhavesh_a_patel{@dell.com}, anngordonross, hlam{@ufl.edu},
jayson.strayer, gaurav.kaul{@intel.com}

Abstract

Field-programmable gate arrays (FPGAs) have largely been used in communication and high-performance computing, and given the recent advances in big data, machine learning and emerging trends in cloud computing (e.g., serverless [1]), FPGAs are increasingly being introduced into these domains (e.g., Microsoft's datacenters [2] and Amazon Web Services [3]). To address these domains' processing needs, recent research has focused on using FPGAs to accelerate workloads, ranging from analytics and machine learning to databases and network function virtualization. In this paper, we present a high-performance FPGA-as-a-microservice (FaaM) architecture for the cloud. We discuss some of the technical challenges and propose several solutions for efficiently integrating FPGAs into virtualized environments. Our case study deploying a multi-threaded, multi-user *compression as a microservice* using FaaM indicates that microservices-based FPGA acceleration can sustain high-performance as compared to a straightforward CPU implementation with minimal to no communication overhead despite the hardware abstraction.

Keywords—FPGA, FPGA-as-a-Service, Compression, Cloud Computing, Container, Virtualization

1. Introduction

With the rapidly increasing demand for cloud computing, there is a corresponding increased interest in using field-programmable gate arrays (FPGAs) to accelerate datacenter workloads. Given an FPGA's computational flexibility, FPGA-based accelerators have been generally applied to applications with intensive, high-performance computing (HPC) demands, achieving orders of magnitude performance improvement and power efficiency as compared to functionally equivalent central processing unit (CPU)-based implementations [4][5][6].

Additionally, an FPGA's reprogramability make FPGA-based accelerators highly suitable for datacenter-wide deployments, especially for workloads that have algorithms that may change over time. However, the economics of scaling new, non-homogenous datacenter architectures combining traditional CPUs with FPGAs remains a significant resource management challenge, which includes deployment, maintainability, and composability across an entire datacenter infrastructure. Addressing these challenges is critical for minimizing operational costs and service downtime in large-scale, production environments.

In spite of this management complexity, the emergence of hyperscale datacenters (i.e., datacenters with high scale-out capabilities) presents an opportunity for accelerator systems that tightly integrate CPUs and FPGAs (e.g., Xeon+FPGA server platform [7]). While these tightly coupled servers enable acceleration of *local* applications that run on each server, to meet performance demands, users must be able to access and distribute applications across a large, global FPGA accelerator pool which shares an optimized communication infrastructure. Finally, for ease of use, this pool must appear as an individual datacenter resource that is accessible to multiple, simultaneous cloud users.

2. Related Work

Recent research efforts explore the flexibility of using FPGAs as hardware accelerators for datacenter services, which are similar to traditional software-based services but realize additional performance benefits.

Byma et al. [8] and Ye et al. [9] proposed integrating FPGA resources into datacenters and the cloud with OpenStack, which is open-source cloud software that uses a hypervisor and virtual machines (VMs). Fahmy et al. [10] introduced a framework that uses a custom resource manager to directly manage virtual FPGA accelerators in the form of partially reconfigurable regions (PRRs).

Cloud-based FPGAs that are used for specific services, such as network function acceleration and deep learning inference, may require low-latency or high-bandwidth communication for streaming data or processing large volumes of data. Caulfield et al. [11] used a layer of FPGAs between the network switches and the servers, providing the FPGAs with direct intercommunication and enabling datacenter-wide acceleration. Ouyang et al. [12] used an FPGA accelerator to enable large-scale deep neural network (DNN) training, and provide online services in a low-cost,

low-power environment.

Despite these advances, integrating FPGAs in the cloud requires the appropriate level of hardware abstraction, as well as FPGA resource management, which is non-trivial. Even with the emergence of new cloud providers, besides Amazon and Microsoft, offering FPGAs to customers, there is still no standard way to make FPGAs more easily available to these users.

In this work, we propose a FPGA-as-a-microservice (FaaM) architecture that presents FPGA accelerators as unique set of datacenter-style services. This architecture allows cloud service providers to offer FPGAs to cloud users in similar ways as CPU and graphics processing unit (GPU) resources are offered, with the added benefit of hardware acceleration and hyperscale.

3. FPGA Microservices

Our approach uses *microservices* (a collection of loosely coupled accelerator services) to offer FPGA accelerators as a collection of shared, lightweight services that scales dynamically with constantly changing datacenter workload demands. Using an FPGA-as-a-microservice (FaaM) architecture for the cloud, FPGA accelerator functionality can be offered as a microservice, enabling application developers to easily leverage many microservice characteristics, including auto-deployment, scalability, dynamic configuration, and disaster recovery [13]. Additionally, since a microservice is stateless, FPGA resources can be quickly provisioned without relying on extra virtualization technology [14], further reducing the time to relocate a microservice in the event of failure.

3.1 Design

In this section, we describe our FaaM design and implementation, which is based on Docker containers. We prototype FaaM using x86-based Xeon+FPGA physical machines running a Linux operating system. We note that

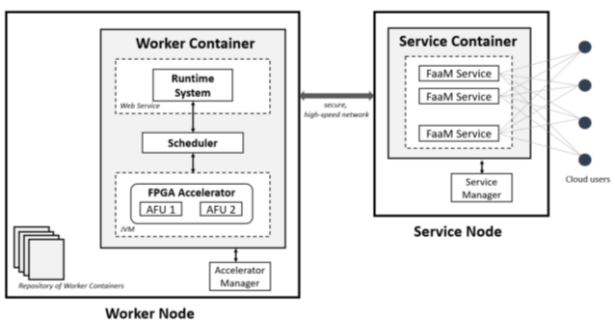

Fig. 1: FaaM architecture

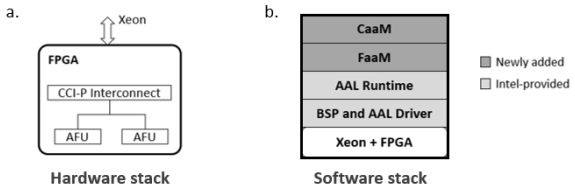

Fig. 2: (a). Basic overview of the Xeon+FPGA hardware stack. (b). Prototyped software stack with Intel-provided components and newly added components. Board Support Package (BSP) and AAL Driver are part of the Intel AAL framework.

the proposed FaaM architecture is not restricted to only Docker containers and Xeon+FPGA platforms, but can be realized for other types of virtualization technologies and FPGA platforms.

Fig. 1 depicts the general FaaM architecture consisting of a *Worker Node* and a *Service Node*. To enable dynamic scaling with increasing datacenter workloads, *Worker Nodes* are decoupled from *Service Nodes*. *Service Nodes* host FaaM services, providing a set of hardware-accelerated functions (e.g., a compression service). FaaM services are deployed in *Service Containers,* simplifying manageability by the FaaM *Service Manager*. To support load balancing, multiple instances of a *Service Container* can be deployed by the FaaM *Service Manager* as a group of identical services, providing fault-tolerant redundancy and scalability. *Service Containers* may also serve unique FaaM services depending on how the FaaM *Service Manager* and the FPGA in the *Worker Nodes* have been configured.

Each *Worker Node* runs a single instance of a FaaM *Accelerator Manager*, which is a separate (privileged) Docker container instance, providing accelerator management functions (e.g., reprogramming the FPGA or providing control and monitoring features). Under the control of the FaaM *Accelerator Manager*, a *Worker Node* hosts one or more *Worker Containers* from a container repository that is accessible by all *Worker Nodes*. Each *Worker Container* abstracts a specific hardware accelerator function (e.g., a compression service), exposing the function as a web service, consequently enabling remote access by *Service Containers*.

A high-speed Ethernet network connects *Service Nodes* with *Worker Nodes*. *Worker Nodes* are behind a secured network, and cloud users have no way of directly interacting with the FPGAs or *Worker Nodes*, except through a set web application programming interfaces (APIs) exposed by *Service Nodes* through *Service Containers*. The APIs are implemented as Java WebSocket, enabling point-to-point inter-node communication.

As shown in Fig. 1, a *Worker Node* is organized into three distinct layers: the FPGA accelerator, the task scheduler, and the Java virtual machine (JVM) runtime system.

FPGA Accelerator: Fig. 2 illustrates the FPGA accelerator layer, where accelerator function units (AFUs) provide the accelerator functionality. The AFUs act as a pool of FPGA configurable resources where different accelerator functions are assigned to each AFU. The accelerator function is constrained in size by the amount of logic resources on the AFU, and must expose a Cache Coherent Interface Protocol (CCI-P) that connects to the CCI-P Interconnect block and to the rest of the components on the FPGA.

Task Scheduler: To schedule cloud users' jobs, we focus on a task scheduler that is local to each *Worker Container*. The role of the scheduler is to admit threads from the web service and schedule these threads on the FPGA. When an accelerator request arrives, the scheduler examines the low-level information from the hardware (such as which AFU is currently unutilized) and makes dispatch decisions that match the corresponding thread to an available AFU. To maintain fair sharing of the AFU, we use a first-come-first-serve (FCFS) scheduling policy and use buffer sizes with minimal overhead, ranging between 32 KB and 128 KB as further discussed in Section 4.2. When an accelerator function is not available, the scheduler defaults to executing the thread on the CPU to maintain acceptable throughput.

JVM Runtime System: We implemented an FPGA runtime system written in Java. The runtime system is designed as a dynamic library shared among multiple threads that can be associated with user requests. We prototype the runtime system atop the Accelerator Abstraction Layer (AAL) software stack provided by Intel. AAL provides low-level accelerator management functionality to the scheduler, allowing the scheduler to call into native C/C++ libraries of AAL. While the FPGA JVM and task scheduler both run as a single JVM process, the web service runs as a separate process, allowing for a different type of web service to be interfaced with the proposed runtime system.

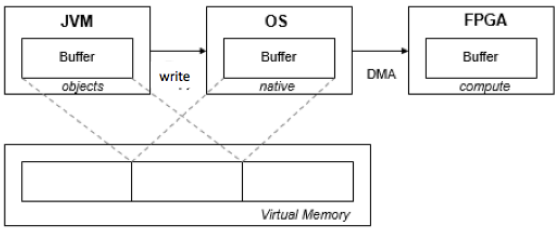

Fig. 3: Data transfer mechanism showing data movement from the JVM and to FPGA memory.

4. Challenges and Solutions

In this section, we present several challenges and solutions when designing the FaaM architecture. To verify the proposed approach, we implement compression [15] as a microservice and evaluate runtime performance as well as overheads.

4.1 Software and FPGA Interaction

While FPGA accelerators are normally manipulated through C/C++ code or low-level libraries, some datacenter-scale applications and frameworks are commonly written in Java - or other runtime-based language such as Scala – running within a (JVM) virtual machine. FPGAs are naturally not supported by JVMs, thus the first step for FPGA-to-application integration is to enable support for the FPGA in the JVM, and bridge the gap between native C/C++ code and the application runtime.

Java Native Interface (JNI) is typically used to address this issue, however JNI does not always deliver an efficient solution. In particular, the cost of moving data between the JVM heap and native memory can adversely impact application performance. Using SWIG (a wrapper and interface generator), we wrote a domain-specific language (DSL) script that automatically generates Java wrappers from native C++ classes. This approach saves us a significant amount time in debugging JNI code directly, while generating clean interfaces that are optimized for our specific native libraries (i.e., the AAL runtime libraries).

4.2 FPGA-to-Host Memory Communication

Since data movement between the JVM and native memory can incur significant overhead, we leverage the non-blocking I/O (i.e., Java NIO) mechanism that is natively built

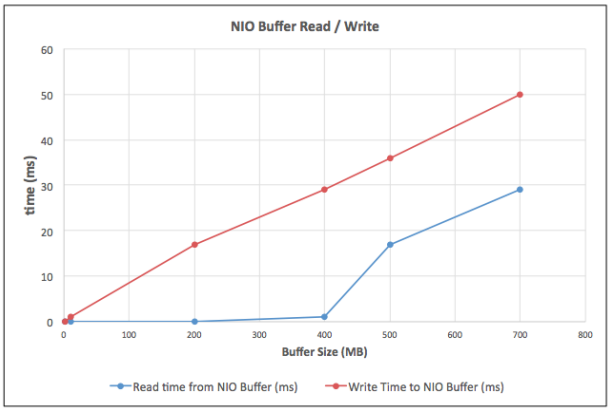

Fig. 4: Buffer read / write performance for different buffer sizes.

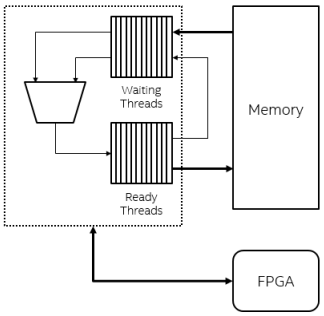

Fig. 5: Multithreading with Java Concurrency: Threads (each fixed with a private buffer size) take turns in offloading compression tasks to the FPGA.

into the Java framework. As shown in Fig. 3, a *buffer* from Java NIO is essentially a block of memory that is wrapped in a Java buffer object. This object is then accessible in Java as a *streaming* Java class, and is free of JVM garbage collection since the underlying memory is outside the JVM heap.

We create NIO buffers of fixed sizes (one per software thread) and re-use individual buffers as many times as the thread associated with a respective buffer is dispatched. Because the allocation of NIO-based buffers can incur overheads (just as with direct memory allocation in C), reusing buffers between non-overlap threads helps to amortize this overhead. As empirically suggested in Fig. 4, we choose buffer size of 64 KB as the optimal transfer size. We also observe that a relatively large amount of time is required by the JVM when establishing large NIO buffers—up to 1ms for buffers as large as 1GB.

4.3 CPU-FPGA Thread Co-existence

FPGAs are naturally high performances, and can rapidly offload CPU threads for compute-intensive tasks such as compression. Therefore, it is necessary to maintain high resource utilization by sharing the FPGA accelerator across multiple CPU threads as illustrated in Fig. 5. To achieve sharing, we implement three versions of an accelerator function interacting with the CPU:

- Single-threaded C++ (ZLIB-FPGA)
- Single-threaded Java (ZLIB-FPGA/JVM)
- Multithreaded Java (ZLIB-FPGA/JVM-T)

We also compare performance with default ZLIB running on a CPU for single-threaded (ZLIB-CPU) and multithreaded (ZLIB-CPU-T) job. While ZLIB-FPGA is a straightforward C++ implementation, ZLIB-FPGA/JVM is

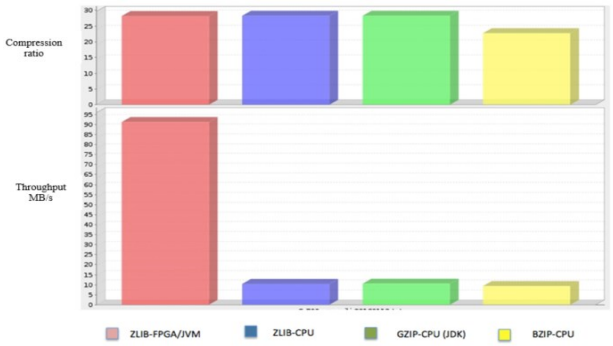

Fig. 6: Single-threaded benchmark of ZLIB-FPGA/JVM versus ZLIB-CPU. Both ZLIB-CPU and GZIP-CPU (JDK) are based on the DEFLATE algorithm, thus their performances are similar. BZIP-CPU has the highest compression ratio but is CPU-intensive.

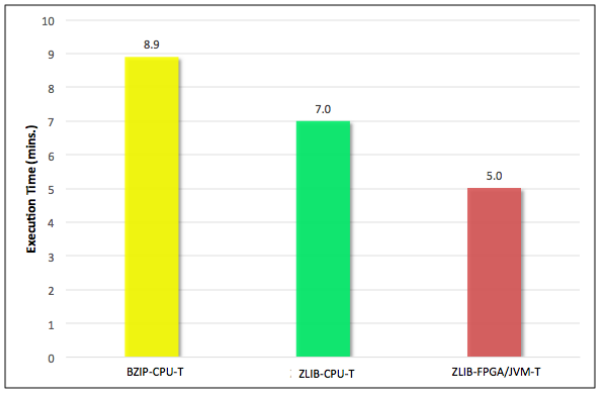

Fig. 7: TeraSort of an 8 GB file in Apache Spark using eight worker threads. "-T" denotes that the compression algorithm is multithreaded.

a wrapper implementation of the former in Java. ZLIB-FPGA/JVM-T is coupled with our task scheduler and together integrated into a multithreaded data processing system (Spark [16]) to demonstrate real world benefits. It is important to note that the purpose of evaluating the performance of ZLIB-FPGA/JVM initially is to ensure that the developed wrapper code meets the performance of the straightforward ZLIB-FPGA implementation at very minimal overheads.

In the single-threaded scenario (Fig. 6), ZLIB-FPGA/JVM shows an average speedup of 9.8X over ZLIB-CPU. This was roughly the same speedup (10X) achieved when comparing the straightforward ZLIB-FPGA implementation with ZLIB-CPU, meaning that the ZLIB-FPGA/JVM has very minimal overhead despite the JVM abstraction.

Having created an efficient JVM version of the FPGA accelerator, we can integrate ZLIB-FPGA/JVM in a multithreaded environment. We use our task scheduler and set the buffer sizes for individual threads to 64 KB (from Fig. 4, 64 KB is the optimal transfer size). The transfer size is also congruent with the chunk sizes on the FPGA accelerator. Moreover, we find that this buffer size is most effective when taking into consideration the Resilient Distributed Datasets (RDD) block size used by the Spark. As shown in Fig. 7 and using our multithreaded JVM implementation, the total application run time is reduced from 7 minutes down to 5 minutes.

4.4 Fault Resilience and Cloud Scaling

An important design factor in a hyperscale cloud is the ability to recover from unforeseen failures and minimize downtimes. For FPGAs deployed in the cloud, this can be particularly challenging due to the setup and initialization steps required. To address this challenge, we extend ZLIB-FPGA/JVM-T and leverage Docker's *GPU passthrough* [17] to create a compression-as-a-microservice (CaaM) framework. The CaaM framework, now exposing ZLIB-FPGA/JVM-T as a containerized service, achieves the same performance as with the non-containerized ZLIB-FPGA/JVM-T implementation.

To provide fault recovery and improve service availability, using the FaaM Accelerator Manager we configure the CaaM framework to automatically restart upon failure, which takes only a fraction of a second as with any standard Docker container that is configured with *Autorestart*.

5. Lessons Learned and Discussions

We presented an architecture for deploying FPGAs in the cloud and highlighted several challenges and solutions for harnessing FPGAs in virtualized environments, such as Docker containers. Motivated by the dynamic nature of datacenter workloads, we proposed an FPGA-as-a-Microservice (FaaM) architecture to allow multiple cloud users to share FPGA accelerator services. Using this FaaM architecture, we implemented compression-as-a-Microservice (CaaM), and demonstrated that FPGA microservices achieve high performance with very minimal runtime overheads.

We summarize the lessons learned as follows:

I. By efficiently designing buffer movement mechanisms between Java's heap memory and native memory used by the FPGA accelerator, it is possible to reduce unnecessary data transfer overheads and achieve acceptable performance that is close to straightforward FPGA implementation in C/C++. Our Java implementation has less than 1% reduction in application performance for the CaaM.

II. Contrary to previous work where a single, shared buffered is created and shared among multiple threads—resulting in thread contentions—our implementations create multiple private non-blocking NIO buffers, resulting in a more efficient computation-to-memory access pattern. By scaling up or down buffer sizes (to a certain threshold) along with the number of threads in relation to the total input work size, a more balanced degree in concurrency (i.e., interleaving) across threads can be achieved. Based on our experimentation, choosing a buffer size that matches the block size of the underlying file system typically results in fewer block misses for data fetched directly from disk.

III. For accelerator service requests, the CaaM framework assumes that the input dataset is domiciled locally on an FPGA-attached node. There is active research to integrate FPGAs with YARN cluster managers, whereby datasets are distributed across multiple nodes (both FPGA- and non-FPGA-attached). With a more aggressive data locality, such cluster manager could subsequently schedule FPGA-specific tasks on the FPGA-attached nodes provided the working sets of the overall data is already locally cached to the nodes.

IV. The fact that FPGA acceleration services implemented using FaaM are encapsulated and isolated across containers, allows container mangers, such as Mesos and Kubernetes, to easily orchestrate such services in a datacenter environment. Future work will include conducting further studies on FaaM with a diverse set of workloads (including machine learning inference) as well as integrating the CaaM framework into streaming applications (e.g., network function virtualization) and data serialization frameworks like Apache Thrift or Microsoft Bond.

References

[1] G. McGrath and P. R. Brenner, "Serverless Computing: Design, Implementation, and Performance," 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW), Atlanta, GA, 2017

[2] B. S. Đorđević, S. P. Jovanović and V. V. Timčenko, "Cloud Computing in Amazon and Microsoft Azure platforms: Performance and service comparison," 2014 22nd Telecommunications Forum Telfor (TELFOR), Belgrade, 2014

[3] https://techcrunch.com/2016/11/30/aws-announces-fpga-instances-for-its-ec2-cloud-computing-service/

[4] E. Ghasemi and P. Chow, "Accelerating Apache Spark Big Data Analysis with FPGAs," 2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld), Toulouse, 2016

[5] E. B. Fernandez, W. A. Najjar, S. Lonardi and J. Villarreal, "Multithreaded FPGA acceleration of DNA sequence mapping," 2012 IEEE Conference on High Performance Extreme Computing, Waltham, MA, 2012

[6] S. Vellas, G. Lentaris, K. Maragos, D. Soudris, Z. Kandylakis and K. Karantzalos, "FPGA acceleration of hyperspectral image processing for high-speed detection applications," 2017 IEEE International Symposium on Circuits and Systems (ISCAS), Baltimore, MD, 2017

[7] Intel Xeon+FPGA. https://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf

[8] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia and P. Chow, "FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack," 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines, Boston, MA, 2014

[9] K. Ye, D. Huang, X. Jiang, H. Chen and S. Wu, "Virtual Machine Based Energy-Efficient Data Center Architecture for Cloud Computing: A Performance Perspective," Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on & Int'l Conference on Cyber, Physical and Social Computing (CPSCom), Hangzhou, 2010

[10] S. A. Fahmy, K. Vipin and S. Shreejith, "Virtualized FPGA Accelerators for Efficient Cloud Computing," 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom), Vancouver, BC, 2015

[11] A. M. Caulfield et al., "A cloud-scale acceleration architecture," 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Taipei, 2016

[12] J. Ouyang, "SDA: Software-defined accelerator for large-scale deep learning system," 2016 International Symposium on VLSI Design, Automation and Test (VLSI-DAT), Hsinchu, Taiwan, 2016

[13] D. Guo, W. Wang, G. Zeng and Z. Wei, "Microservices Architecture Based Cloudware Deployment Platform for Service Computing," 2016 IEEE Symposium on Service-Oriented System Engineering (SOSE), Oxford, 2016

[14] H. Jin, "Virtualization Technology for Computing System: Opportunities and Challenges," 2008 10th IEEE International Conference on High Performance Computing and Communications, Dalian, 2008

[15] Zlib. http://zlib.net/

[16] Apache Spark https://github.com/apache/spark

[17] J. P. Walters et al., "GPU Passthrough Performance: A Comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL Applications," 2014 IEEE 7th International Conference on Cloud Computing, Anchorage, AK, 2014

[18] D. Liu and L. Zhao, "The research and implementation of cloud computing platform based on Docker," 2014 11th International Computer Conference on Wavelet Actiev Media Technology and Information Processing(ICCWAMTIP), Chengdu, 2014

[19] D. Jaramillo, D. V. Nguyen and R. Smart, "Leveraging microservices architecture by using Docker technology," SoutheastCon 2016, Norfolk, VA, 2016

[20] 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), Beijing, 2015

[21] Chen, Fei, Yi Shan, Yu Zhang, et al. 'Enabling FPGAs in the Cloud', Proceedings of the 11th ACM Conference on Computing Frontiers, (2014)

[22] R. Perrey and M. Lycett, "Service-oriented architecture," 2003 Symposium on Applications and the Internet Workshops, 2003. Proceedings., 2003

[23] D. G. Puranik, D. C. Feiock and J. H. Hill, "Real-Time Monitoring using AJAX and WebSockets," 2013 20th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS), Scottsdale, AZ, 2013