FPGAVirt: A Novel Virtualization Framework for FPGAs in the Cloud

Joel Mandebi Mbongue, Festus Hategekimana, Danielle Tchuinkou Kwadjo, David Andrews, and Christophe Bobda
Computer Science and Computer Engineering Department
University of Arkansas, Fayetteville, Arkansas
Email: {jmmandeb, fhategek, dtchuink, dandrews, cbobda}@uark.edu

***Abstract*—Field-Programmable Gate Arrays (FPGAs) are becoming important components within commercially available cloud computing systems. However, the FPGAs are not yet sufficiently abstracted within existing software ecosystems. Contrary to how applications are transparently scheduled across general purpose processors, software processes need to explicitly provision and control communications with hardware circuits within the FPGAs. In this paper, we introduce a novel virtualization framework called FPGAVirt that leverages Virtio to implement an efficient communication scheme between virtual machines and the FPGAs. FPGAVirt avoids the overhead of context switches between virtual machine and host address spaces by using the in-kernel network stack for transferring packets to FPGAs. Experimental results show FPGAVirt can deliver an additional 2$\times$ to 35$\times$ performance increase compared to current state of the art virtualization approaches.**

***Keywords*-Cloud; Virtualization; FPGA; Overlay; Virtio-Vsock**

I. Introduction

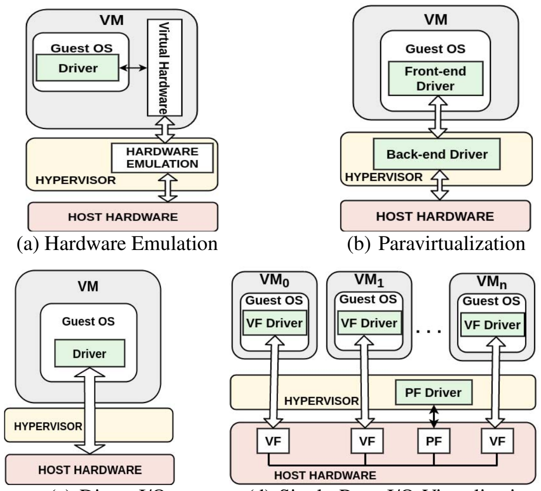
Field-Programmable Gate Arrays (FPGA) are gaining interest in general purpose computing systems due to their ability to serve as energy efficient domain customizable accelerators [1]. Amazon is now exposing FPGAs to application developers in their cloud based EC2 F1 instances. Baidu is providing the same type of FPGA services within their cloud infrastructure [2]. While these systems introduce application programmers to the energy and performance benefits of FPGAs, additional research is required to transparently manage the FPGA within the virtual machine software stacks that run in todays cloud based systems. Specifically, to achieve this capability new approaches are needed to address the following challenges: **(i) Abstraction:** FPGAs must be seen by the cloud management system as a resource that can be requested, assigned and deallocated. **(ii) Sharing:** while the programming model used in general purpose processors allows sharing execution time intervals between VMs' processes, the operation model of FPGAs imposes a restriction: VMs must be able to access and program specific regions of the FPGA. **(iii) Isolation:** each VM should execute in an environment totally isolated from other ones. Moreover, the whole cloud infrastructure must be protected from malicious applications that users could introduce. Allowing tenants to program FPGAs in the cloud therefore requires the adoption of security mechanisms to prevent hardware tasks from accessing or tampering resources not belonging to their domain. **(iv) Availability:** in practice, it ensures that data and services are available as much as possible. In the context of provisioning FPGAs, architectural supports should be provided to make sure that users would always access their hardware designs.

The contribution of this paper are: (1) A hardware/software co-design framework that allows VMs to access virtualized FPGAs. (2) An overlay dividing the physical FPGA into virtual functions. The overlay provides isolation to hardware tasks using hardware sandboxes (HWSB) [3], and employs configurable communication channels. The latter allows the dynamic allocation of additional resources to users.

II. Related Work

Recent work focusing on FPGAs in the cloud such as [4], [5], and [6] propose virtualization frameworks. [4] presents a virtual FPGA model that does not allow users to program FPGAs with their own designs, rather it relies on precompiled hardware accelerators. Moreover, no communication link is enabled between hardware tasks mapped on virtual FPGAs belonging to the same user. Consequently, data will therefore be copied back and forth through the VM address space. Finally, the employed FPGA virtualization mechanism imposes either a modification of the host operating system (HOS), or high overhead due to the address translation between VM and Host address spaces. As opposed to [4], [5] introduces an approach where users access FPGA resources over a network and deploy their hardware designs. It divides a single physical FPGA into virtual FPGAs. Similarly to [4], [5] does not enable communication channels among virtual FPGAs. This restriction prevents user from expanding their designs as the size of virtual FPGAs is fixed. Furthermore, no support protects the whole infrastructure from hardware malicious applications that could either access or modify the content of other applications, and eventually attack the cloud infrastructure. The work presented in [6] also suffers from the same limitations observed in [5]. In response to these shortcomings, we design a model in which FPGA's regions allocated to VMs can communicate, and only access resources belonging to their domain. Virtualization techniques can be categorized as software-based approaches and hardware-based ones. Emulation and paravirtualization

2159-6190/18/$31.00 ©2018 IEEE
DOI 10.1109/CLOUD.2018.00122



(a) Hardware Emulation (b) Paravirtualization (c) Direct I/O (d) Single Root I/O Virtualization

Figure 1. Hardware emulated by the Hypervisor (1a). A front-end driver in the guest directly communicates with a back-end diver in the host (1b). VMs directly access the hardware (1c). VMs share the same physical hardware (1d).

fall into the software-based category. For the first one, the hypervisor emulates hardware devices (see Figure1a). VMs can then use native drivers, resulting in no modification of guest operating systems (GOS). The major drawback is the overhead of context switches between GOS and HOS. Paravirtualizing hardware (see Figure1b) offers a more efficient scheme as a front-end driver in the VM is connected to a back-end driver in the host to remove context switches' overhead. The two previous methods insert a hypervisor between VMs and the hardware, introducing penalties in VM I/O performance. Hardware-based virtualization techniques aim to mitigate that overhead. Direct I/O (see Figure1c) gives VMs a direct access to physical hardware, resulting in better performance compared to previously mentioned methods. Unfortunately, this approach does allow to share physical devices among VMs, which is one of the most important requirement to virtualizing hardware in the cloud. Specifications like Single Root I/O Virtualization (SR-IOV) allow multiple VMs to access the same physical device (see Figure1d). It features several Physical Functions (PF), each accessible through Virtual Functions (VF). Each VM can directly reach a VF to achieve bare metal performance [7]. Though the latter offers some sharing capability, each VF can only be accessed by one VM, making this approach not scalable in a cloud-oriented deployment. There is therefore a need for an hybrid approach, combining both the best of software-based with hardware-based virtualization techniques to share FPGA resources among VMs in the cloud.

In this work, we propose a paravirtualized virtio-based framework that opens communication channels between VMs and FPGAs. Further, we design an FPGA overlay featuring VFs purposed to run guest's hardware tasks by leveraging partial reconfiguration (PR). The framework proposed removes the tight VM-to-VF coupling observed in SR-IOV to allow runtime reassignment of FPGA resources. The overlay also implements HWSBs to prevent hardware tasks from accessing resources out of their domain.

III. Background

A. Virtio

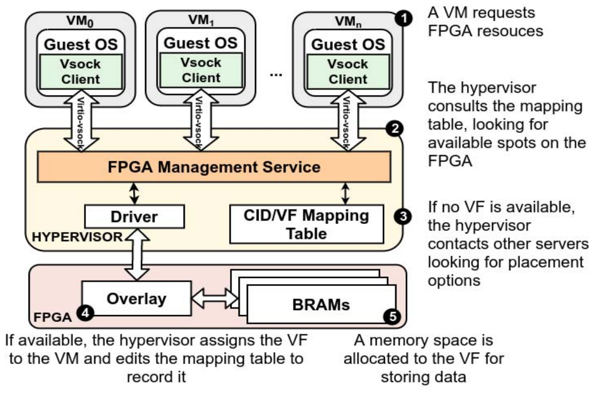
Virtio is an I/O virtualization framework that was initially implemented by Rusty Russel in 2008 for Linux environments [8]. It is an abstraction layer providing communication interfaces between guests and host in paravirtualized environments. Its implementation called virtio-serial, by enabling the host with the ability to expose multiple serial ports, allows character devices in the guest to stream data.

B. Virtio-Vsock

Virtio-serial communication model presents some limitations. First of all, serial ports are built to support one-to-one connections; only one guest will be able to correspond with the host at a time. In an environment with several VMs trying to access a host, implementing a many-to-one connection will then require adding an extra arbitration layer or opening several virtio ports. Another limitation comes from the fact that serial ports only provides stream semantics. This implies that applications relying on datagrams will need an additional transformation layer to exchange with the host. To overcome these boundaries, [9] presents Virtio-Vsock, a new communication framework devised from virtio. It leverages the socket API to remove the need for guest applications to handle character devices, and enables many-to-one communications. Instead of going through the network, prone to connection shortages due to factors like firewall configurations, it takes advantage of AF_VSOCK Unix sockets. The major change introduced by this socket family resides in the addressing scheme. Addresses are made of a context identifier (CID) combined with a port number, as opposed to IP addresses and ports in TCP/IP and UDP sockets. Additionally, it supports both stream and datagram semantics. The host always keeps a well known CID while each guest is assigned a unique CID at boot time. The main advantage of this framework lies in the fact that it is not emulated. It integrates a vhost driver (Vhost_Vsock in Figure 2) into the host to move packets using the Linux kernel network stack like vhost-net[1] [9].

Figure 2. Communication Scheme

[1]In-kernel guests networking performance enhancement.



VM0
VM1
VMn
Guest OS
Vsock Client
Virtio-vsock
1 A VM requests FPGA resouces
2 The hypervisor consults the mapping table, looking for available spots on the FPGA
3 If no VF is available, the hypervisor contacts other servers looking for placement options
FPGA Management Service
Driver
CID/VF Mapping Table
HYPERVISOR
Overlay
BRAMs
FPGA
4 If available, the hypervisor assigns the VF to the VM and edits the mapping table to record it
5 A memory space is allocated to the VF for storing data

Figure 3. Overall Framework

IV. FPGAVirt: Our FPGA Virtualization Framework

A. Infrastructure

The proposed framework is a hardware/software co-design that uses Virtio-vsock for communications between VMs and FPGAs. Figure 3 highlights its main components and operation. The *FPGA Management Service* controls the access to the physical FPGA. It uses the *CID/VF Mapping Table* to keep track of FPGA regions assigned to VMs. The FPGA mainly contains an overlay architecture provisioning VFs to VMs, and a set of Block RAMs (BRAMs) to store VFs' data. FPGA resources being finite, VFs are designed with a specific size. This implies that each design provided by users in the cloud should match that size requirement. Users will therefore split designs which are bigger than a VF into smaller hardware tasks. The overlay architecture makes it possible to assign several VFs on the same FPGA or on distant FPGAs to a user. This allows to meet the availability requirement for cloud services as hardware tasks belonging to a user can be distributed over several FPGAs. Upon assigning a VF to a VM, the *FPGA Management Service* updates the *CID/VF Mapping Table* with the goal of recording the FPGA usage. Once done, it configures the HWSB, making sure to block any attempt to access a VF or BRAM which do not belong to the same user. HWSBs respond to the isolation requirement given that each user will only access resources belonging to his domain. The possibility of allocating VFs to users at runtime fulfills the sharing requirement. A channel is opened with FPGA regions as long as they belong to the domain of a VM. Finally, VFs provide an abstraction of FPGAs to cloud users. Each VM can be assigned several VFs on demand, and use partial reconfiguration to implement specific hardware tasks.

B. FPGA Overlay

The architecture of the overlay is a Torus interconnect [10] consisting of a two-dimensional array of routers and programmable processing elements (respectively R and PE in Figure 4). PEs communicate through a flexible and scalable network-on-chip providing a high path diversity and minimal routes. Figure 4 presents an instance of the overlay using 2×2 routers, each surrounded by four PEs. The number of PEs attached to each router, and how many routers to use can be modified depending on resources available on the chip. Routers control paths taken by data to make sure that each packet reaches its destination, and implement the *First Come, First Served* scheduling. A PE consists of two components: a VF and a HWSB. The HWSB is a module programmed by the hypervisor when a VF is assigned to a VM. It has two main functions. Firstly, it prevents from unauthorized communications. A VF can only communicate with VFs belonging to the same VM. Secondly, it configures distant communications. When a distant communication is initiated, the HWSB adds the identifier of the distant VF to the packet to allow the routing. Each VF is a PR region that is allocated to a VM to implement hardware tasks.

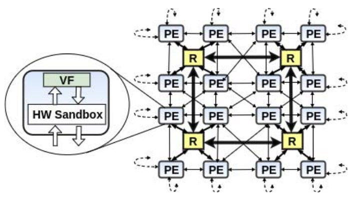


Figure 4. Overlay Architecture

V. Experimental Results

In this section, we present preliminary results from prototyping our framework. For testing purposes, we implemented the proposed architecture on a server running CentOS-7 with a kernel of version 4.10.0. It was equipped with an Intel Core i7 CPU embedding 12 cores of frequency 3.50GHz. The server had 32GB of RAM and we used QEMU version 2.11.50 for emulating VMs. Each VM was running an Ubuntu 16.04.01 operating system with 4GB of RAM. We implemented the FPGA overlay in VHDL on a Stratix V board using Quartus Prime Standard Edition 16.0.0. The FPGA board was finally attached to the server through a PCIe Gen3 $\times 16$ connector. The focus of this preliminary work being on the overall infrastructure, we evaluate VM-to-VF round trip times (i.e from VM to FPGA registers, and from FPGA registers to VM) for various payload sizes. We do not discuss execution time of hardware tasks on FPGAs, as it depends on the designs that users will program in the VFs. Instead, we analyze read and write times in the FPGA registers to evaluate the effectiveness of our framework. We additionally investigate the impact of increasing the number of VMs on the FPGA access time. Finally, we provide some comparisons with previous works and discuss how the overlay occupies FPGA resources.

Figure 5 studies how both increasing the size of the payload and the number of VMs running concurrently impacts the access time to FPGAs. The first observation is that generally, as the number of CPU cores assigned to VMs increases, the longer it takes to access the FPGA regardless

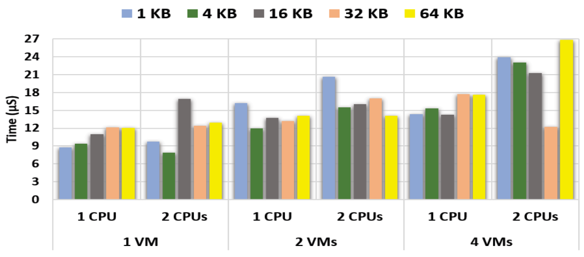


Figure 5. Average Round Trip Times

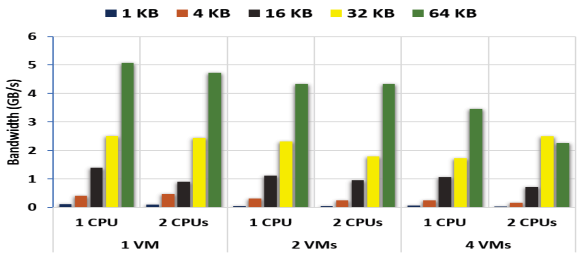


Figure 6. Average Bandwidth

of the size of the payload. This is imputable to the overhead introduced by QEMU that emulates virtual CPUs running VMs in addition to the scheduling taking place in the HOS: assigning more resources to VMs generally tend to slow down the host computer. Looking at average round trip times, it oscillates between 8 and 26μs which is smaller than the average 600μs in [5] (about 35$\times$ slower), and is far from the minimum minute needed to provision FPGAs in [6]. [4] reported about 15μs for one VM to perform a round trip with a payload of size 4KB, which is about twice bigger than the 8μs we obtain with one VM for the same payload size. This difference can be explained by the efficent Virtio-Vsock communication scheme that avoids context switches. Figure 6 examines how the bandwidth scales with the number of VMs running. Typically, the bandwidth diminishes as we increase the number of VMs. This is simply a resultant of several VMs trying to transfer payloads of different sizes to their VFs simultaneously. The bandwidth goes down from about 5GB/s with one VM (running on 1 CPU) to 2.3GB/s with four VMs (running on 2 CPUs each) for 64KB payloads. It is also observed that smaller payloads do not significantly influence the average bandwidth independently of the number of VMs or virtual CPU cores. Table I summarizes the FPGA usage for an overlay of 2×2 routers, each with 4 PEs. It demonstrates that routers and HWSBs are not resource hungry as the design only used about 1% of the FPGA logic resources. The overlay design achieved a f_{max} of 227MHz. In the end, the overall size of the overlay on the FPGA will vary depending on resources allocated to PR regions implementing VFs.

Table I
FPGA RESOURCE USAGE

Resource	Usage (Used/Total)
ALM	3305 / 234720 (1%)
Total Pins	84 / 1064 (8%)

VI. Conclusion

This work presented a new virtualization framework for FPGAs in the cloud. It exploits concepts from paravirtualization and SR-IOV to propose a flexible and effective model. It differs from SR-IOV by allowing runtime remapping of VFs to VMs, and uses Virtio-Vsock for fast communications between VMs and host. Future work includes studies to decrease the overhead introduced by the hypervisor in the management of FPGAs.

References

[1] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A performance and energy comparison of fpgas, gpus, and multicores for sliding-window applications," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. ACM, 2012, pp. 47–56.

[2] S. Gianelli, "Baidu deploys xilinx fpgas in new public cloud acceleration services," https://www.xilinx.com/news/press/2017/baidu-deploys-xilinx-fpgas-in-new-public-cloud-acceleration-services.html, 2017.

[3] C. Bobda, J. Mead, T. J. Whitaker, C. Kamhoua, and K. Kwiat, "Hardware sandboxing: A novel defense paradigm against hardware trojans in systems on chip," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2017, pp. 47–59.

[4] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang, "Enabling fpgas in the cloud," in *Proceedings of the 11th ACM Conference on Computing Frontiers*. ACM, 2014, p. 3.

[5] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow, "Fpgas in the cloud: Booting virtualized hardware accelerators with openstack," in *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*. IEEE, 2014, pp. 109–116.

[6] N. Tarafdar, N. Eskandari, T. Lin, and P. Chow, "Designing for fpgas in the cloud," *IEEE Design & Test*, 2017.

[7] B. Zhang, X. Wang, R. Lai, L. Yang, Y. Luo, X. Li, and Z. Wang, "A survey on i/o virtualization and optimization," in *ChinaGrid Conference (ChinaGrid), 2010 Fifth Annual*. IEEE, 2010, pp. 117–123.

[8] R. Russell, "virtio: towards a de-facto standard for virtual i/o devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, 2008.

[9] S. Hajnoczi, "Virtio-vsock:zero-configuration host/guest communication," https://www.linux-kvm.org/page/KVM_Forum_2015, 2015.

[10] J. Mandebi Mbongue, D. Tchuinkou Kwadjo, and C. Bobda, "Flexitask: A flexible fpga overlay for efficient multitasking," in *Proceedings of the ACM Great Lakes Symposium on VLSI, GLSVLSI*. ACM, 2018.