A PCIe DMA engine to support the virtualization of 40 Gbps FPGA-accelerated network appliances

Jose Fernando Zazo*, Sergio Lopez-Buedo*†, Yury Audzevich‡, Andrew W. Moore‡
*NAUDIT HPCN
Calle Faraday 7, 28049 Madrid, Spain
†High-Performance Computing and Networking Research Group, Universidad Autonoma de Madrid
Ciudad Universitaria de Cantoblanco, 28049 Madrid, Spain
‡Computer Laboratory, University of Cambridge
15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom

Abstract*—Network Function Virtualization (NFV) allows creating specialized network appliances out of general-purpose computing equipment (servers, storage, and switches). In this paper we present a PCIe DMA engine that allows boosting the performance of virtual network appliances by using FPGA accelerators. Two key technologies are demonstrated, SR-IOV and PCI Passthrough. Using these two technologies, a single FPGA board can accelerate several virtual software appliances. The final goal is, in an NFV scenario, to substitute conventional Ethernet NICs by networking FPGA boards (such as NetFPGA SUME). The advantage of this approach is that FPGAs can very efficiently implement many networking tasks, thus boosting the performance of virtual networking appliances. The SR-IOV capable PCIe DMA engine presented in this work, as well as its associated driver, are key elements in achieving this goal of using FPGA networking boards instead of conventional NICs. Both DMA engine and driver are open source, and target the Xilinx 7-Series and UltraScale PCIe Gen3 endpoint. The design has been tested on a NetFPGA SUME board, offering transfer rates reaching 50 Gb/s for bulk transmissions. By taking advantage of SR-IOV and PCI Passthrough technologies, our DMA engine provides transfers rate well above 40 Gb/s for data transmissions from the FPGA to a virtual machine. We have also identified the bottlenecks in the use of virtualized FPGA accelerators caused by reductions in the maximum read request size and maximum payload PCIe parameters. Finally, the DMA engine presented in this paper is a very compact design, using just 2% of a Xilinx Virtex-7 XC7VX690T device.

***Index Terms*—Network Function Virtualization, Virtual Network Appliance, FPGA-based acceleration, SR-IOV, PCI Passthrough, PCIe, DMA engine, NetFPGA SUME**

I. Introduction

At the moment there is a huge interest in Network Function Virtualization (NFV) technologies. The advantages of moving from proprietary hardware appliances to virtualized software appliances are well known: reduced costs and power, reduced time to market and service deployment, possibility of using a single platform for different applications and/or users, and, it encourages openness and innovation [1]. Additionally, NFV complements well Software-Defined Networking (SDN): an optimal flexibility is achieved when everything is software-based, and the virtualization infrastructure can be used to provide a virtual machine within which to execute the control plane.

The goal of NFV is to substitute proprietary, ASIC-based, network appliances with software programs running on a virtual machine. Although ASICs definitely provide the best performance, there have been a number of developments that allow removing the bottlenecks in a configuration based commodity Network Interface Cards (NICs) and an off-the-shelf servers in the recent years. Specifically, libraries and drivers for fast packet processing such as DPDK [2] allow bypassing the burdensome networking stack of the operating system. Moreover, technologies such as PCI passthrough and SR-IOV allow improving the performance of virtual machines so that the overhead of virtualization is minimized [3].

The advantage of FPGAs is that they can provide the speed of hardware and the flexibility of software. That is, one can take advantage of ASIC-like hardware acceleration without losing the reprogrammability of software. This is especially interesting in the NFV environment: one could think of having FPGA-accelerated virtual networking appliances, composed by a software program and an FPGA bitstream. In this scenario, conventional NICs would be substituted by commodity FPGA boards with on-board networking interfaces (such as NetFPGA). Therefore, a virtual network appliance would be deployed by launching a virtual machine in a server and reconfiguring an FPGA-based networking board. FPGA configuration could be total or partial; the latter case corresponds to the one where several virtual network appliances share the same board.

One of the difficulties of this FPGA-accelerated approach for virtual networking appliances is the communication between the software running on a virtual machine and the FPGA design itself. Fortunately, the techniques developed for conventional NICs, also apply here; these are PCI passthrough and SR-IOV. However, implementing a PCIe DMA engine in an FPGA logic with these capabilities is far from trivial. In this paper, we present an open-source solution capable of achieving data transfer rates well above 40 Gbps for a FPGA to virtual machine transmission. These transfer rates are achieved by using PCI passthrough and SR-IOV as well as a huge-page, zero copy driver. This design has been developed as a part of NetFPGA-SUME project [4].

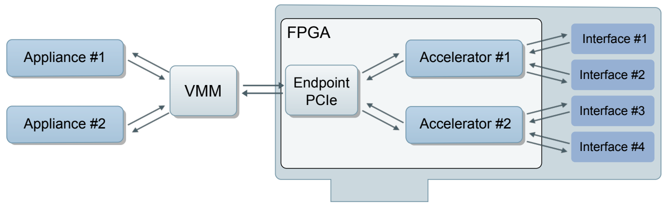

Fig. 1: Example of FPGA-accelerated firewall: Physical connections

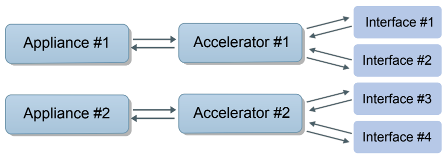

Fig. 2: Example of FPGA-accelerated firewall: Logical connections

II. Usage model

We envisage adding commodity FPGA boards to servers in order to improve the packet processing capabilities of virtualized software appliances. The FPGA boards will also feature network interfaces, so these boards replace the conventional Ethernet NICs.

We consider future virtual network functions to be composed of a software program and an FPGA design. Packets will arrive to the FPGA, where they will be processed and either forwarded to the other network interface inside the board or sent to the CPU for further processing in software (SW). This is a flexible computing model, which allows network applications to be completely run in SW, hardware (HW), or a mix of both. Fig. 1 shows an example, where two virtual FPGA-accelerated firewalls are running in one server. The FPGA board has 4 interfaces with each pair of them being dedicated to one virtual firewall. The FPGA device is split into two virtual accelerators, each running one instance of the HW accelerator for the firewall. On the other side, the CPU runs two virtual machines, each running one instance of the SW program for the firewall appliance. Communication between the SW running in the virtual machines and the virtual HW accelerators will be made via the PCIe bus of the commodity FPGA board.

A major challenge in this approach is on establishing an efficient communication between virtual machines and virtual hardware accelerators. Two key technologies allow designers to overcome this obstacle: SR-IOV and PCI passthrough. SR-IOV enables the creation of several virtual functions in an FPGA, each associated to one virtual machine in the CPU. PCI passthrough allows for a direct access of the virtual machine to the PCI bus, without the intervention of the host operating system. Using these technologies, the logical model for the two virtual FPGA-accelerated firewalls is shown in Fig. 2. The software running on a virtual machine sees a direct connection to its virtual FPGA accelerator, without any interference from the host operating system or the other virtual appliance.

III. Related Work

References [5] and [6] provide a comprehensive summary of hardware acceleration techniques for NFV applications. However, the approach proposed in these papers differs from the tightly coupled FPGA accelerator model considered in this work. On the one hand, Kachris et al. propose in [6] an FPGA-based NFV platform where network functions are completely implemented in FPGA. On the other hand, Nobach et al. in [5] present Elastic AH, an approach based on pools of software virtualized network functions (VNFs) and acceleration hardware modules (AHs).

A similar model to that followed in this work (tightly coupled FPGA accelerators) is presented in [7]. Although it makes reference to the use of PCIe SR-IOV technology, no implementation details are provided. The efficacy of FPGA-based accelerated solutions in a medium-scale industrial deployement has been also shown by Microsoft in [8], [9]. Certain papeps in algorithm acceleration research domain complement NFV field. [10] details how to communicate an FPGA accelerator with a virtual machine using PCI passthrough technology. [11] divides the FPGA device into several areas, each dedicated to implement a virtual FPGA accelerator, and partial runtime reconfiguration is used to change one virtual coprocessor without disturbing the others. In [12] several virtual processing units are created out of a single FPGA board by using a time-multiplexed scheme.

Finally, [13] shows how to integrate FPGA accelerators in a cloud for NFV. Although this paper follows the model of isolated FPGA-based functions, the proposed methodology could be easily ported to the tightly-coupled accelerator scheme being considered in our work.

IV. Design overview

In order to provide a complete PCIe solution, both hardware and software components need to be implemented. On the hardware side, the DMA core should be implemented in FPGA logic. On the software side, driver and user intefacing libraries are needed. The latter implement the required APIs utilized by the user-level applications in order to communicate to the FPGA board.

The key elements to enable efficient operation of the PCIe solution in a virtualized environment are the SR-IOV and PCI Passthrough technologies. Firstly, SR-IOV inherits the use of an IOMMU from the Direct I/O technology, which allows guest VMs to directly use peripheral devices through DMA and interrupt remapping. This is what is referred to as PCI Passthrough. Secondly, SR-IOV offers the possibility of creating multiple virtual devices, also known as Virtual Functions (VFs). One or many VFs are associated to a Physical Function (PF). A PF is a full-featured PCIe function, whereas a VF lacks of full configuration resources. A VF is meant to be plugged directly to a guest VM, thus ensuring that a misconfiguration of the device cannot be done, since the full configuration space is only available in the PF, which is always managed by the host operating system.

A. DMA core

The Xilinx 7 Series Gen3 Integrated Block for PCI Express [14] takes care of the lower layers (physical and data link) of the PCIe communication, and also of the PCI configuration space. However, the user needs to implement the transaction layer in the FPGA logic. Fortunately, only three different types of TLPs (transaction layer packets) need to be considered: Memory Read Request, Memory Write Request and Completion with Data.

The interface of the integrated PCIe endpoint to the FPGA logic is essentially organized as four AXI4-Stream buses, along with many other control signals. These buses are: Completer Request (CQ), Completer Completion (CC), Requester Request (RQ), and Requester Completion (RC). The requester interfaces are used when the FPGA acts as a master in DMA transfers: RQ for sending memory read and write requests to the host, and RC for receiving the completions corresponding to the previous read requests. On the contrary, the completer interfaces correspond to the slave interface of the FPGA, typically used when the host reads or writes any of the control registers implemented in the programmable logic. The CQ interface is used to receive memory read and write requests from the host, and the result of a read request should be sent back to the host via the CC interface as a completion TLP.

Finally, the PCIe endpoint also provides a mechanism for the FPGA logic to generate interrupts at the host. Our DMA core uses MSI-x interrupts as a recommended solution for SR-IOV environments (by Xilinx). In addition, MSI-X scheme allows extra flexibility with larger number of IRQs natively supported (up to 2048); there are enough interrupts available even if many concurrent VMs are being used simultaneously.

Before entering into details, two key concepts that need to be explained are:

1) A descriptor is a data record containing all the information needed to perform a DMA operation. That is, a descriptor should at least contain the host address where data will be transferred, size of the transfer and a boolean value indicating if an interrupt will be generated on completion. In our DMA core, descriptors are stored in FPGA memory, which is mapped in the host memory space via the corresponding BAR register of the PCIe endpoint.
2) An engine is an unidirectional DMA block, which can operate in the D2H (device to host) or H2D (host to device) direction (mutually exclusive). In our design, engines contain a set of descriptors, organized as a circular list. Each engine has a FIFO where data is consumed/stored by the FPGA application that makes use of the DMA core.

For a D2H operation, the host first fills one or many DMA descriptors. Once descriptors are ready, the host asserts the enable bit in the DMA core to start the transfer. The transfer will be splitted into several Memory Write Request TLPs, so that the negotiated maximum payload (generally 64-256 bytes) is not exceeded. No feedback from the host is expected, since PCIe uses posted writes. Additionally, PCIe credits should be taken into account. Credits are the basis for flow control in PCIe. The PCIe endpoint announces the number of credits available, and when credits are running low, the DMA core should stop sending TLPs.

For a H2D transfer, the host also fills one or many DMA descriptors, and the sets the enable bit to start the operation. However, in this case the DMA core sends Memory Read Request TLPs to the host, and waits for its answer in form of Completion With Data TLPs. The requested size is limited by the negotiated maximum read request (128-4096 bytes). In order to boost performance, the DMA core keeps sending read requests without waiting for completions. The first difficulty here is that responses from the host typically contain less bytes than requested (for example, a 4096-byte read request might be answered with 16 completions, each sized 256 bytes). However, the main challenge is that completions may arrive unsorted. That is, a completion for request i may arrive after a completion for request $i+1$. Fortunately, all completions for request i arrive in a ascending address order.

A window mechanism has been implemented in order to cope with unsorted completions. Requests are sent until the specified window is exhausted or run out of PCIe credits. Completions are stored in FIFOs, one FIFO for each read request (each read request corresponds to an element in the window). When all completions for the first request have arrived, the window can advance one position. The Tag field of PCIe TLP is used to identify to which request the completion corresponds. In our design, the window size is parameterizable.

B. SR-IOV

SR-IOV is supported by the PCIe endpoint of the FPGA, being possible to create up to 2 physical functions (PFs) and 6 virtual functions (VFs). In this work a total of 1 PF and 1 VF were defined.

Routing of TLPs for the completer interface is carried by the Bus number:Device number:Function number (BDF) notation, with BDF being 8 bits/5 bits/3 bits correspondigly. The bus number is constant, so that a total of 8 bits are available. The two possible PFs are represented by values 0x00 and 0x01 in the pair Device:Function, while values from 0x40 to 0x45 correspond to VFs. This way the core can distinguish the source/destination of a TLP in the completer interfaces. In the requester interfaces there is no such feature, so the tag is used as the device identifier. The tag has a width of 8 bits, which are used for redirecting between the PFs and VFs, and for sorting TLPs in a H2D operation.

The overall architecture is shown in Fig. 3. In the DMA SR-IOV block, components are replicated for each VF. For the completer logic, the Bus:Device:Function field is used to route the request to the right VF.

C. Physical and Virtual driver

The physical and the virtual driver are practically analogous in behaviour. Nevertheless, the physical driver is the only

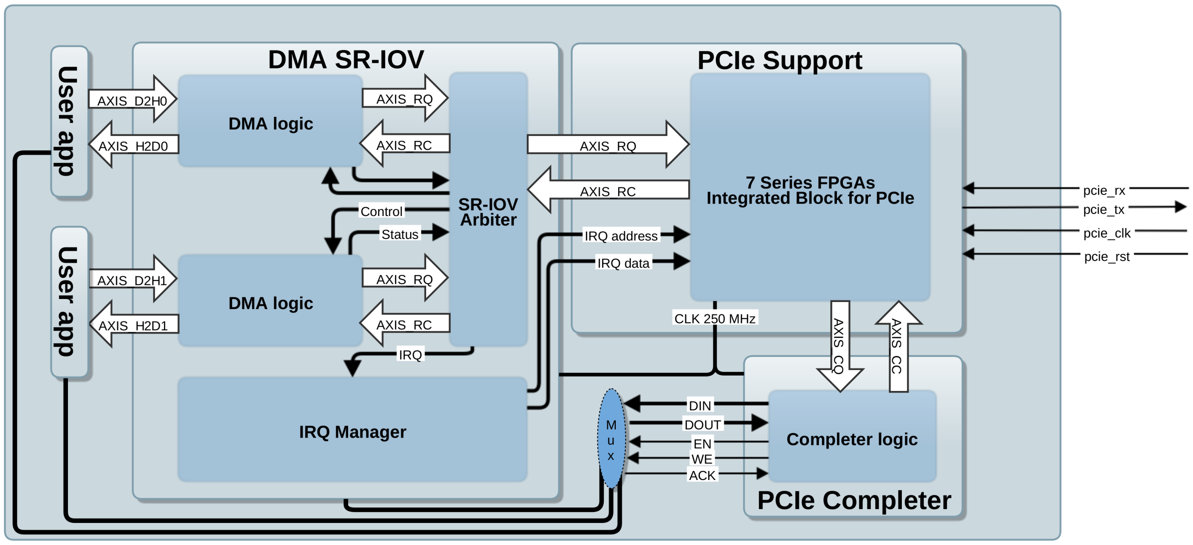

Fig. 3: Block diagram of the PCIe DMA engine

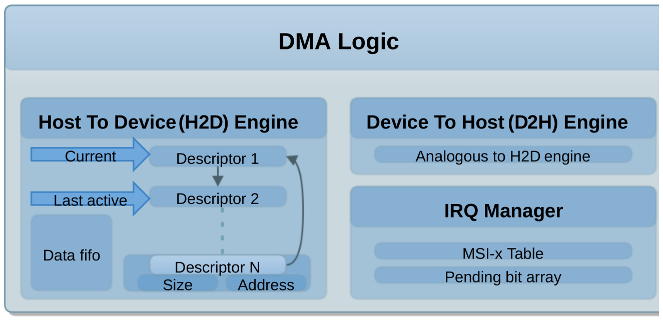

Fig. 4: Hardware level implementation of DMA logic

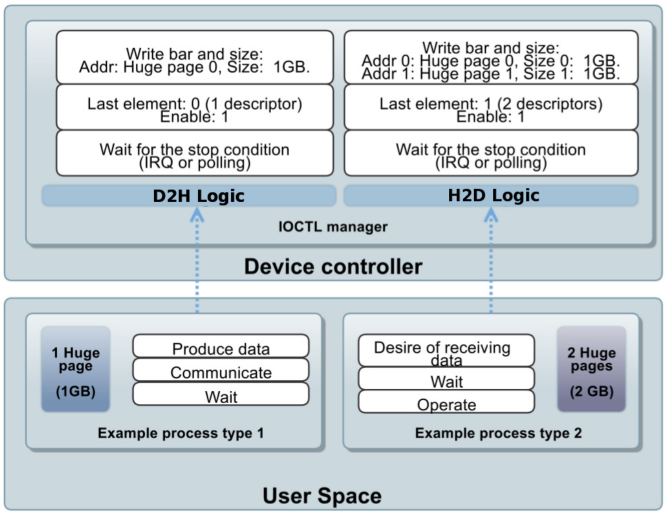

Fig. 5: Interfacing of the DMA core from User Space

one capable of accessing the PF resources, enabling VFs an setting the maximum payloads and read requests. Both driver can attend IRQs and support direct accesses of the CPU to th device and DMA operations.

As it was mentioned before, every VF has its own DMA logic, Fig. 4. That is to say: Data FIFOs, a MSI-x table vector and a pending bit array (PBA) table, and a ring of descriptors for every engine (by default one H2D and one D2H engine). When a data buffer needs to be transferred, typically as many descriptors as non-contiguous regions of memory will be filled. Each descriptor contains physical address (64 bits), size (64 bits) and a boolean value indicating if an IRQ is to be generated by the end of the operation. On top of that, the number of descriptors and a enable bit will have to be asserted in order to start the DMA operation. In the Fig. 5 this process is detailed, omitting the VM layer. A classical producer-consumer pattern is followed:

- If a D2H operation is required, the software running at user space sends to the driver the buffer pointer and the quantity of data to receive. The device controller obtains the physical address corresponding to the supplied pointer. Note that if the user space buffer is mapped to a non-contiguous area in physical memory, an array of physical addresses will be obtained, so it will be necessary to configure as many descriptors as elements in this array. After the DMA core has been configured, the thread can wait. When data has been delivered by the DMA core to the host, an interrupt awakes the driver thread, which signals the ending of the operation to the user. The DMA core is abstracted to the user, so aspects such as IRQ handler or descriptors are completely managed in kernel space.
- When the involved direction is H2D, the buffer pointer and quantity of data to be transmitted is similarly provided to the driver. Once again, physical addresses are obtained and as many descriptors are configured as non-contiguous regions of physical memory exists. When

this operation ends, the driver notifies the user space application, so the buffer can be reused.

Finally, the use of huge pages is highly advisable in order to allow bigger memory regions to be transferred with only one descriptor.

D. Interaction with user space

A middleware layer hides all low-level details to the user. Communication with the driver is encapsulated in 4 different IOCTL commands: H2D_DMA, D2H_DMA, H2D_CPU and D2H_CPU. The H2D_DMA and D2H_DMA commands set a DMA transfer, and the H2D_CPU and D2H_CPU commands are used to respectively write and read configuration registers. Each IOCTL receives an additional argument where the required information is provided. BAR, address and a pointer to a DWORD are the parameters needed by H2D_CPU and D2H_CPU. For the H2D_DMA and D2H_DMA commands, the parameters are buffer pointer, size, generated IRQ and enable bit. The enable bit lets the user configure several descriptors before data transfer begins.

At compile time, polling or interrupt-based operation can be selected. In case of polling, The H2D_DMA and D2H_DMA commands are blocking and the driver performs a busy wait. However, for the interrupt-based operation, the H2D_DMA and D2H_DMA commands are non-blocking, and there is an additional IOCTL (PENDING_DATA) to check if the operation has already finished.

E. Experimental tests

For testing in the H2D direction, the host fills with integers a buffer implemented in huge pages, and programs a DMA transfer to send it to the FPGA. The hardware design checks the received values to make sure that no ”gaps” between the consecutive pieces of data have been detected. The experiment is reversed for the D2H direction: the hardware generates the integers whilst the user space application checks the received values for correctness.

In the tests with virtualized environments, one VM and one VF are created using the KVM hypervisor. The main reason for choosing KVM is its robustness and openness to the community. Additionally, most Linux distributions offer support for the KVM kernel module nowadays, so the experiments could be easily ported to other platforms. For the tests, the resources allocated to the VM are 8GB of RAM memory and a maximum of 4 CPU cores. VT-x, VT-d and SR-IOV capabilities are enabled at both BIOS and host OS levels.

The chosen architecture for the experiments consists on a Supermicro X9DRD-iF motherboard, dual Intel Xeon CPU E5-2650 v2 @ 2.60GHz microprocessors, 64 GiB of DDR3 RAM clocked at 1600 MHz (8 banks of 8 GiB, 4 modules connected to each CPU) and a NetFPGA SUME board (featuring a Virtex-7 XC7VX690T-2FFG1761C FPGA) directly attached to the first CPU. Thread affinity is used in order to guarantee that all tasks are mapped to the first CPU and that the second CPU is completely idle.

Slice logic	Endpoint PCIe	DMA Logic	DMA Core	SR-IOV Core
FF	3334	5320	8784 (1.03%)	8951 (1.05%)
LUTs	3810	5141	9171 (2.13%)	9238 (2.15%)
Memory LUTs	56	0	136 (0.08%)	136 (0.08%)
Block RAMs	9	8	19 (1.29%)	19 (1.29%)
BUFGs	5	0	5 (15.62%)	5 (15.62%)

TABLE I: Device Utilization Summary

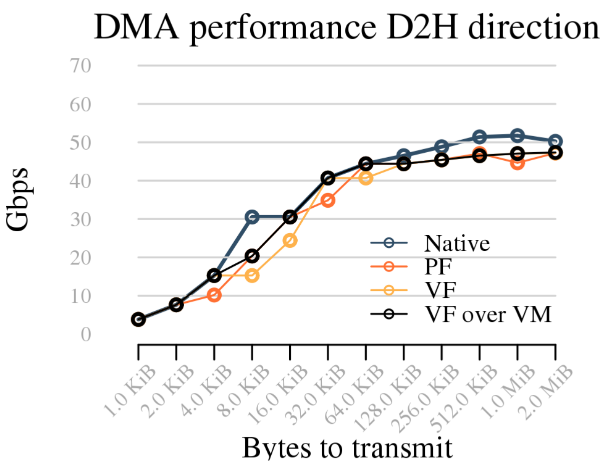

Fig. 6: Performance for D2H transferences

V. Results

The device utilization is summarized in Table I. The results of the experiment are plotted in Fig. 6 and Fig. 7. Four basic cases have been considered: DMA transactions with a hardware design where no SR-IOV support is provided (maximum payload MP equals 256B), DMA transactions over the PF (MP equals 128B), DMA transactions over the VF (attached to the host machine) and DMA transactions over the VF (attached to a VM).

In the native case (no SR-IOV), the experiments show a clear trend: the overhead for small data transfers is significantly high. Actually, a good DMA performance is only obtained for transfers bigger than 256 KB. In the D2H direction a maximum of 51.74 Gbps is obtained when copying a region of 1 MB and in the H2D, a maximum of 50.40 Gbps has been measured. These measures include the time needed to configure the descriptors, so that is the reason why a bigger performance is not obtained (the theoretical maximum for PCIe is around 57 Gbps for 256-byte payloads).

For the SR-IOV cases, performance in the D2H direction is at least 90% of that of the native case for transfers bigger than 128 KB. For obvious reasons, if multiple VMs were implemented, the available bandwidth would be shared among them.

However, in the H2D direction the results are less promising: a VM using the VF only achieves a maximum of 33.67

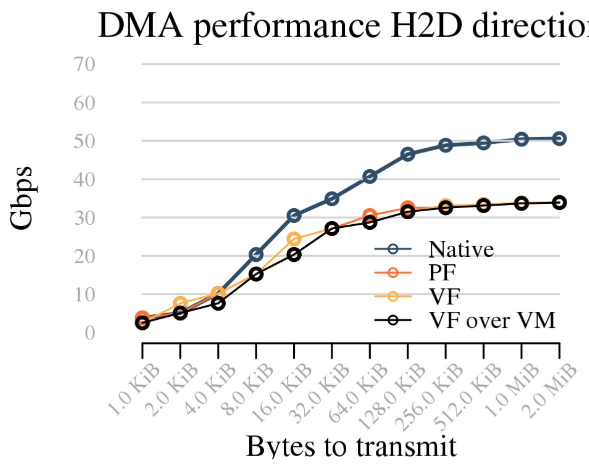

Fig. 7: Performance for H2D transferences

Gbps (66.80% of the native case). This can be explained by the fact that in the selected platform, the SR-IOV support limits the maximum read request size and maximum payload parameters of the virtual devices to 128 bytes only. In the native case, the maximum payload size available is 256 bytes while the maximum read request is 4096 bytes. The huge decrease in the maximum read request parameter, from 4096 to 128 bytes, is the reason for the poor performance; the identical transfer will require 32 times more read request TLPs.

VI. Conclusions

In this paper we have presented the design of a PCIe DMA engine with full support for virtualization (SR-IOV) and capable of transferring data from the FPGA to the host at rates greater than 40 Gbps. This block is the key element to enable FPGA acceleration of virtual network appliances. The final goal of this work is to have an NFV architecture where conventional NICs are substituted by FPGA boards, in order to overcome the limitations at very high rates (40+ Gb/s) of software-only implementations.

The benefit of the proposed design is that it is a fully scalable block, capable of creating several virtual functions, each with one or more DMA engines. As a result, very complex functions can be created in the FPGA. The obtained results show that the overhead of implementing SR-IOV virtualization for the device to host transfers is minimal and the main cause for reduced throughput is the configured value of maximum read request size in the host to device direction. Note that host to device communications are non-posted, so all the components associated to the management of completions play a crucial role. Additionally, the design is very compact, occupying just 2% of the selected device for the minimal configuration.

This core is being developed as a part of NetFPGA SUME project and, as an open-source design, the goal is to foster research in FPGA-accelerated NFV. As future work we envision the integration of this DMA engine with a framework for the dynamic reconfiguration of the FPGA, in order to support several virtual accelerators in one FPGA, that can be changed at run-time. This way, the virtualization of networking appliances would be complete: both at software and at hardware acceleration levels.

Acknowledgment

This work was partially supported by the Spanish Ministry of Economy and Competitiveness under the project PackTrack (TEC2012-33754) and by the European Union through the Integrated Project (IP) IDEALIST under grant agreement FP7-317999. The stay of Sergio Lopez-Buedo at the University of Cambridge was funded by the Spanish Government under a ”Jose Castillejo” grant. Additionally, this research was sponsored by EU Horizon 2020 SSICLOPS (agreement No. 644866) research program and EPSRC through Networks as a Service (NaaS) (EP/K034723/1) project.

References

[1] European Telecom Standards Institute (ETSI), “Network functions virtualisation introductory white paper,” October 2012.
[2] Intel. Intel DPDK: Data plane development kit. [Online]. Available: http://dpdk.org/
[3] IBM. Linux virtualization and pci passthrough. [Online]. Available: http://www.ibm.com/developerworks/library/l-pci-passthrough/
[4] N. Zilberman, Y. Audzevich, G. Covington, and A. Moore, “NetFPGA SUME: Toward 100 Gbps as Research Commodity,” *Micro, IEEE*, vol. 34, no. 5, pp. 32–41, Sept 2014.
[5] L. Nobach and D. Hausheer, “Open, elastic provisioning of hardware acceleration in nfv environments,” in *2015 International Conference and Workshops on Networked Systems (NetSys)*, March 2015, pp. 1–5.
[6] C. Kachris, G. Sirakoulis, and D. Soudris, “Network Function Virtualization based on FPGAs: A Framework for all-Programmable network devices,” June 2014. [Online]. Available: http://arxiv.org/abs/1406.0309
[7] X. Ge et al., “OpenANFV: Accelerating Network Function Virtualization with a Consolidated Framework in Openstack,” in *Proceedings of the ACM Conference on SIGCOMM*, August 2014, pp. 353–354.
[8] A. Putnam et al., “A reconfigurable fabric for accelerating large-scale datacenter services,” in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, June 2014, pp. 13–24.
[9] A. Greenberg, “SDN for the Cloud,” keynote speach, at the annual ACM SIGCOMM conference on the applications, technologies, architectures, and protocols for computer communication, August 2015 [Accessed: 2015 11 02]. [Online]. Available: http://conferences.sigcomm.org/sigcomm/2015/pdf/papers/p0.pdf
[10] W. Wang, M. Bolic, and J. Parri, “pvFPGA: Accessing an FPGA-based hardware accelerator in a paravirtualized environment,” in *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Sept 2013, pp. 1–9.
[11] E. El-Araby, I. Gonzalez, and T. El-Ghazawi, “Virtualizing and sharing reconfigurable resources in high-performance reconfigurable computing systems,” in *Second International Workshop on High-Performance Reconfigurable Computing Technology and Applications, 2008.*, Nov 2008, pp. 1–8.
[12] I. Gonzalez, S. Lopez-Buedo, G. Sutter, D. Sanchez-Roman, F. J. Gomez-Arribas, and J. Aracil, “Virtualization of reconfigurable coprocessors in hprc systems with multicore architecture,” *J. Syst. Archit.*, vol. 58, no. 6-7, pp. 247–256, Jun. 2012.
[13] S. Byma, J. Steffan, H. Bannazadeh, A. Leon-Garcia, and P. Chow, “Fpgas in the cloud: Booting virtualized hardware accelerators with openstack,” in *IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2014, pp. 109–116.
[14] Xilinx, “Virtex-7 FPGA Gen3 Integrated Block for PCI Express v4.0 LogiCORE IP Product Guide 023, Vivado Design Suite,” July 2015.