2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC) | 978-1-6654-3860-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/MCSOC51149.2021.00012

Host Bypassing: Direct Data Piping from the Network to the Hardware Accelerator

Ralf Kundel*, Kadir Eryigit*†, Jonas Markussen¶‡, Carsten Griwodz‡, Osama Abboud§, Rhaban Hark*, Ralf Steinmetz*
*Multimedia Communications Lab, Technical University of Darmstadt, Germany
ralf.kundel@kom.tu-darmstadt.de
†Zoi TechCon GmbH, Stuttgart, Germany
‡Department of Informatics, University of Oslo, Norway
¶Dolphin Interconnect Solutions AS, Oslo, Norway
§Huawei Technologies, Munich, Germany

***Abstract*—Computer networks have become very important and influential over the last years for many common services such as Internet connectivity as well as time-sensitive applications such as videotelephony. Furthermore, approaches like in-network computing enable the offloading of latency-critical and high-performance network functions into the network, *e.g.* 5G network functions, to enable such time-sensitive applications.**

In this work, we show how FPGAs in PCIe-based systems, which are typically used as hardware accelerators for latency-critical in-network functions, can be integrated into the data path. Our approach, named *host bypassing*, allows direct data transfer from the network interface to the accelerator and accomplishes substantial performance benefits over existing state-of-the-art approaches. Our detailed evaluation results demonstrate the possibility of achieving deterministic low latency while operating under heavy load without any packet loss. In addition, fewer CPU resources are required.

***Index Terms*—PCIe, FPGA, Offloading, Bypassing, DPDK**

I. Introduction

The continuously growing number and scale of digital services in the Internet and all underlying networks has led to numerous challenges for network and data center operators. First, due to the very high demand for digital services, a lot of computing power is needed at the lowest possible energy consumption. Second, high flexibility and scalability are required in networks in order to meet the continuously changing needs of on-top applications [1].

While CPUs are highly flexible and capable of serving arbitrary computing applications, most network switches and network interface cards (NICs) are realized with non-programmable Application Specific Integrated Circuits (ASICs). Since ASICs can only provide fixed and limited functionality, the Network Function Virtualization (NFV) [2] paradigm has resulted in a movement of more networking functionality onto the CPU for flexibility reasons. NFV describes the execution of network functions in virtualized software environments, demanding flexibility that is typically achieved by executing them on standard x86 or ARM servers and not on purpose-built devices. However, this reduces the overall performance [2].

Many recently arisen applications have strict Quality of Service (QoS) requirements, including the requirement of time-critical processing within the data path. 5G networks, in particular, impose extreme demands in terms of latency, throughput and jitter while moving increasingly towards softwarized networks. 5G O-RAN systems are examples that require low latency, high throughput and lower jitter between their Radio Units and Distributed Unit [3]. This gives rise to a need for flexible hardware acceleration within the data path.

While achieving these QoS requirements is inhibited by today's common approach to NFV, which sacrifices performance to gain flexibility, in-network computing approaches focus on bringing the functionality back from servers into the data path of the network. Combining the ideas of NFV and hardware acceleration for in-network computing opens up huge potentials to fulfill the constraints of time-critical networking functions. However, integrating programmable hardware accelerators, such as Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs), into the data path is challenging. In the context of commodity servers with a standard PCIe infrastructure, moving network packets between the NIC and an accelerator requires copying data via system memory, which introduces additional latency in the data path.

To illustrate this, we introduce a simple reference architecture for a computer system with an FPGA as a hardware accelerator in Figure 1. State-of-the-art approaches store incoming data from the network first in the main memory (DRAM). The incoming data is then transferred via Direct Memory Access (DMA) from the main memory into the FPGA in a second step. After being processed by the accelerated network function implemented on the FPGA, named f_x, the data is sent out via the same path. The data is moved at least four times within the system, and CPU interaction is needed.

In this work, we present `host bypassing` to drive commodity NICs with an FPGA directly as shown in Figure 1. It reduces the number of memory copies to two that do not require CPU involvement. As a consequence, we expect an increased throughput, lower latency and strongly reduced CPU utilization. This approach can be realized with any poll-mode

978-1-6654-3860-5/21/$31.00 ©2021 IEEE
DOI 10.1109/MCSoC51149.2021.00012

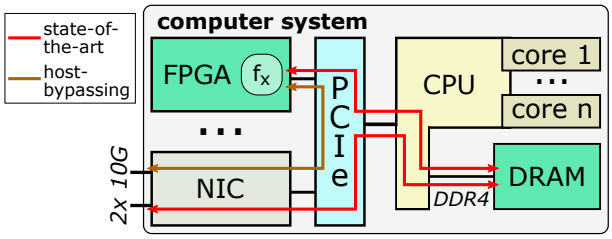

Fig. 1. Reference computer system with a PCIe-based FPGA accelerator. The red path represents the current state of the art for packet I/O to accelerators. The host bypassing approach utilizes PCIe peer-to-peer capabilities.

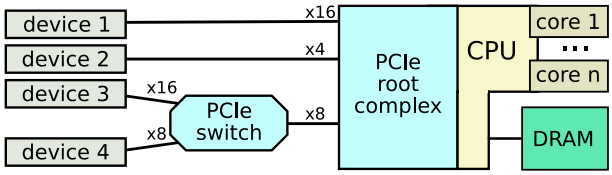

Fig. 2. Example of a PCIe subsystem topology in a computer system. Several devices are connected to the CPU through the root complex. Devices may read from or write to system memory (DRAM) using DMA.

capable hardware and no special NIC is needed. In addition, it can be extended to chain several hardware-accelerated network functions within the same PCIe domain [4].

The outline of this paper is as follows: First, the fundamental basics of the PCIe subsystem and NIC drivers are introduced. Second, we present the design and implementation details of the host bypassing approach. Third, we evaluate and discuss the performance characteristics of the presented approach in several scenarios. Finally, after discussing related works of other researchers, we conclude this paper.

II. Background

This section introduces the two main technologies underlying on our work: 1) the PCIe bus standard for integrating peripheral devices and hardware accelerators into computer systems, and 2) user space poll mode drivers for NICs.

A. PCIe Subsystem

PCIe is the de facto standard for high-speed computer expansion that connects hardware devices such as NICs, GPUs and FPGAs to a computer system [5]. Figure 2 depicts an example of a PCIe subsystem topology: Several devices are connected to the CPU, either directly or via a PCIe switch on the motherboard. Each connection is a point-to-point link, consisting of 1 to 16 lanes. Each lane is a full-duplex serial connection. Data is striped across multiple lanes, so broader links yield higher bandwidth. Connecting the CPU to the PCIe subsystem is the so-called "root complex". Devices are mapped into the same physical address space as the CPU, and memory transactions are routed in the PCIe subsystem based on these mapped addresses. Because of this mapping, CPU applications can access device memory in the same way as system memory (DRAM). Likewise, if a device is capable of

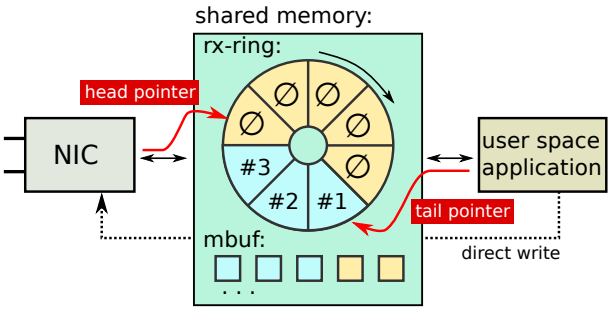

Fig. 3. DPDK principle of receiving incoming packets directly into the user space. Synchronization between the application and NIC is realized by a descriptor ring and a shared memory buffer.

DMA, it can directly read from and write to the main system memory.

A device may even access other devices in the PCIe subsystem directly, as they are mapped into the same address space. This is called "peer-to-peer" in PCIe terminology. PCIe switches are assigned the combined address range of their downstream devices, allowing memory transactions to be routed over the shortest path in the subsystem instead of passing through the root complex. In Figure 2, transactions between device 3 and device 4 would be routed directly through the switch without involvement of the root complex.

B. User Space Poll Mode Driver

The idea of driving NICs directly from user space has become very popular over the last years, as this provides several benefits over kernel based approaches. In contrast to the whole Linux kernel stack, only the required functionality is implemented in the application in a simplistic manner [6]. In addition, zero-copy mechanisms are possible, which means that the arriving packets are stored in a memory region where the application can process them. By that, huge performance benefits can be achieved.

In this work, we build upon the Data Plane Development Kit (DPDK) library, a user space driver supporting a wide range of NICs [7]. The main principle of DPDK is using ring buffers to exchange packets between NIC and application, as shown in Figure 3 for receiving packets. In a first step, the NIC reads from a free descriptor ring entry indicated by the head pointer. This entry contains the physical memory address where the next received packet should be stored. As soon as the packet arrives, the NIC writes it to this memory address. Last, packet metadata and the information that a packet was received are written into the descriptor ring and the head pointer is advanced by one. The user space application polls the descriptor ring at the tail pointer location asynchronously to the NIC for new packets. When the application has read a packet, the rx tail pointer is advanced by one and written into the NIC. This tail pointer indicates the range of free descriptors to the NIC. In Figure 3, the NIC currently receives three packets but are not yet processed by the application.

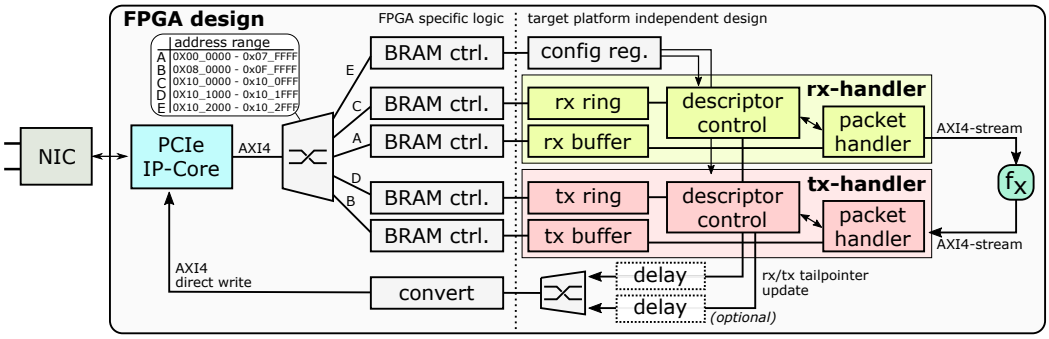

Fig. 4. FPGA internal design for host bypassing. The left side is FPGA type specific while the right side is platform independent. Modules in light green depict the receive logic, while red modules represent the transmission part of the DPDK driver. Compare Section II-B for ring and buffer functionality.

III. Design and Implementation

The overall design can be divided into several modules within the FPGA, as shown in Figure 4. In the following subsections, we first focus on the PCIe data path that maps the ring memory structures and packet buffer memory into the FPGA. Second, we describe the software driver stack for initializing the NIC and FPGA. In the third subsection, the design of the NIC driver within the FPGA is explained.

A. Shared Memory Mapping into the FPGA

The host bypassing approach presented in this work operates with commodity poll-mode capable NICs. Consequently, the same behavior as the software driver is emulated by the FPGA. As introduced in Section II-B, current user space poll-mode drivers communicate with applications through a shared memory region in the main memory of the host system. For the FPGA implementation, four shared memory regions, two descriptor rings and a memory buffer, are realized within the FPGA and accessible via DMA by the NIC. Figure 4 shows the implementation of these data structures in the FPGA. The two descriptor rings and two memory regions for receiving and sending packets are realized with internal SRAM-based block RAM (BRAM) memory cells. In addition, a fifth module with a BRAM interface was specified for configuration purposes. Together, these five memory blocks provide a simple interface to the BRAM controller. The PCIe module, realized by an Intellectual Property Core (IP-core) of the FPGA vendor, is connected by a crossbar to the five memory controllers, and the module forwards read and write requests according to the mapping table shown in Figure 4. In the implementation for Xilinx FPGAs, we used the AXI4 data bus with a data width of 256 bit and running at $250MHz$, supporting a theoretical symmetric throughput of $64Gbit/s$ within the FPGA. The PCIe module was configured to Gen. 3 and 8 lanes, providing a symmetric throughput of up to $64Gbit/s$. As the NIC used in our prototype has a link speed of $10Gbit/s$, this memory-mapped design should not cause any bottlenecks.

As the block memories create an abstraction layer between the platform independent NIC driver and the FPGA-specific logic on the PCIe-side, our open-source prototype implementation[1] supports Xilinx and Intel FPGAs. For that, only modifications of the FPGA specific logic are needed. Note that an IP-core for PCIe, memory interconnect and BRAM controllers are available for both vendors.

In addition to the memory-mapped I/O, it is also necessary to update both tail pointers in the NIC as described in Section II-B. For that, a low-throughput write channel from the rx and tx logic to the PCIe module is used.

B. Software Driver Modifications

Our prototype is based on the DPDK library, which required minor changes to work with the host bypassing approach. We added the functionality to provide a custom physical memory address for the rx and tx descriptor rings. In addition, we created access to the DPDK device wrapper by the newly added function *rte_eth_dev *eth_dev_get(uint16_t port_id);*.

In the FPGA driver, only one function is required to initialize the configuration register of the FPGA with 1) the physical base address of the NIC, 2) the physical base address of the FPGA and 3) the start command. Finally, a kernel module is loaded for the FPGA whose only purpose is to make the physical memory address regions available. After this, all required functionality is implemented on the FPGA.

C. FPGA-based NIC Driver

After having mapped the shared memory regions into the internal BRAM of the FPGA successfully, the next step is the implementation of the DPDK driver functionality within the FPGA. As the rx and tx logic can be treated independently, implementing these two directions is also done independently from each other. The driver logic has therefore been split up into two modules:

rx-handler: This module is connected to the rx descriptor ring and the rx packet buffer on the PCIe side. Towards the application, it provides the received packets through a standardized AMBA AXI4-stream interface.

tx-handler: Mirroring the rx-handler, this module receives packets on an AXI4-stream interface from the network function running on the FPGA and sends it out via PCIe. It is connected to the tx descriptor ring and tx packet buffer.

[1]https://github.com/ralfkundel/HostBypassing

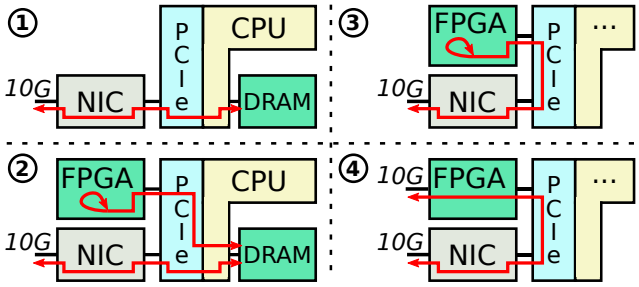

Fig. 5. Evaluation scenarios for host bypassing. Scenario 1+2: baseline/state-of-the-art, scenario 3+4: bidirectional/unidirectional bypassing.

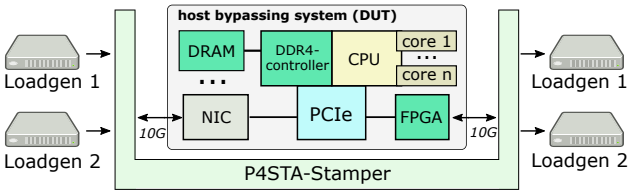

Fig. 6. Testbed setup for generating test traffic and measuring QoS characteristics. Packets can be injected and received either by a $10Gbit/s$ link to the commodity NIC or the FPGA. The FPGA port is only used for measuring single-direction delays. The stamper device of the P4STA setup can create fine-shaped test loads and measure latency and loss with very high accuracy.

Besides having access to the shared memory rings and buffers, both handlers write directly into the NIC's physical address range in order to update the rx and tx tail pointers.

The rx-handler polls the rx descriptor ring continuously for new packets. While busy waiting would use the capacity of an entire CPU core, this is done by the FPGA with little computational effort. As soon as a new packet has been received, the rx handler reads it out from the rx buffer and sends it out on the AXI4-stream interface. The descriptor control of the rx-handler then increases the rx tail pointer and writes an empty rx buffer memory address into the descriptor ring to allow the NIC to receive a new packet on that descriptor. Last, the descriptor control starts polling the next descriptor ring entry.

The tx path works analogously to this. The packets are first stored in an empty region of the tx buffer. Second, the tx-handler updates the tx ring entry accordingly. Third, the descriptor control of the tx-handler updates the tail pointer in the NIC to indicate the packet to be sent.

Updating the rx and tx tail pointer is a 4 byte write access on the NIC via the direct write data path. Even though this is comparatively little data compared to network packets (of up to 1514 bytes), it is still an independent PCIe bus access with a corresponding overhead. In order to decrease this overhead, multiple tail pointer updates are batched together by the optional delay modules. This module updates the tail pointer either after some time or after a number of packets to be sent, *e.g.*, $2500ns$ or 8 packets.

IV. Evaluation

In the following, we demonstrate the performance of the host bypassing approach by presenting several performance characteristics. Note that while the results presented here are from our implementation using Xilinx Alveo U50 FPGAs, experiments with Intel Stratix 10 FGPAs yield similar results. For that, we compare two baseline scenarios (scenario 1+2) with two host bypassing scenarios (scenario 3+4) as shown in Figure 5.

In the first scenario, we consider the performance of a DPDK application running in the user space without any hardware acceleration. In this scenario, the data is transferred only between the NIC and the main memory of the CPU. The second scenario realizes packet I/O over the main memory of the CPU and copies the data for processing to the FPGA. These two scenarios represent the state of the art of high-performance packet processing without/with hardware accelerators. In the third scenario, all incoming and outgoing packets are transmitted directly between NIC and FPGA via PCIe without going through the main memory and CPU.

As the used FPGA also provides a native Ethernet port, we can evaluate sending and receiving over PCIe separately. Therefore, we include a fourth scenario where we use this Ethernet port to send out packets received by the NIC and copied via PCIe to the FPGA. Incoming packets on the Ethernet port of the FPGA are sent by the NIC via PCIe. While this is similar to the third scenario, as packets can be sent and received over PCIe directly from the NIC, this last scenario allows the investigation of one-way delay measurements.

The descriptor size was 64 on the FPGA and did not affect the performance. In software, 256 descriptor ring entries are used as this has shown up the best performance.

A. Testbed Setup

For gathering performance evaluation results, we built upon the existing open-source framework P4STA for benchmarking and validating network functions [8]. In the following, we consider the host bypassing implementation as Device under Test (DUT), directly connected to the P4STA stamper as shown in Figure 6. Depending on the evaluation scenario, packets can be injected and received either on the NIC port or directly from the FPGA via a $10Gbit/s$ port. Test packets are created by the load generators and are aggregated, counted and timestamped within the stamper. In addition, packets can be duplicated in order to generate very high loads. This setup allows the detection of one single lost packet in a multi-million packet test and latency measurements with only a few nanoseconds error. Furthermore, packet reordering can be easily detected. All tests are performed with UDP test packets of $300bytes$-size and $10Gbit/s$ load, except the tests that are explicitly denoted differently. Even though packets in computer networks typically have a size of close to the Maximum Transfer Unit (MTU), e.g. 1500 bytes, with smaller packets we can stresstest the system as the $packets/s$ rate increases at a constant link speed. Furthermore, latency-critical applications typically have much smaller average packet sizes [9].

All measurement results are latency corrected by the measurement overhead of the setup and the length of the used

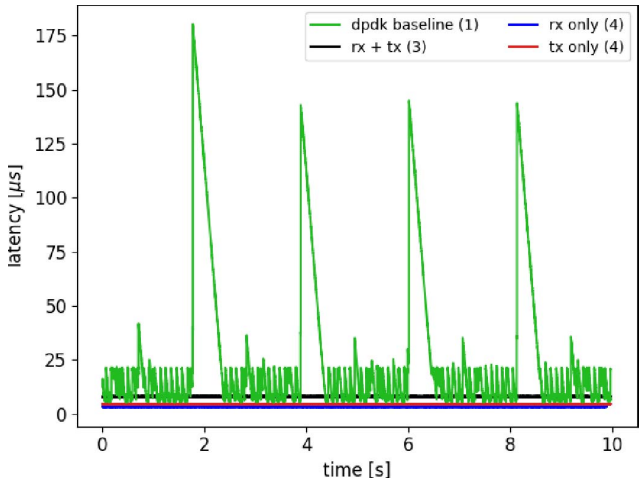

Fig. 7. Measured latency over time for the DPDK baseline, receiving +sending packets via PCIe, receiving only and sending only via PCIe.

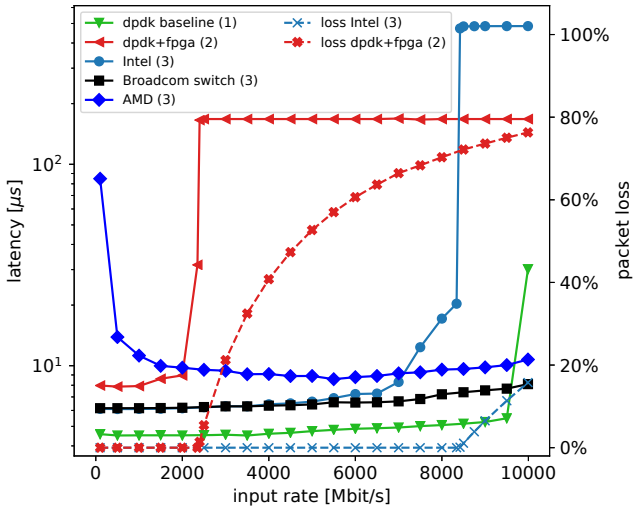

Fig. 8. Observed packet loss and latency for three different PCIe root/switch architectures. The DPDK baseline, Broadcom switch and AMD root complex scenarios did not show up any packet loss.

fibers. By that, all results show the port-to-port latency only.

B. General Forwarding Behavior

As mentioned in Section II-A, PCIe devices can be either attached directly to the CPU root complex or an external PCIe switch. In the following, we will consider only the PCIe-topology of both PCIe devices, NIC and FPGA, are connected to a PCIe switch of the type Broadcom PEX8747. The CPU root complex is not involved in the data path, and in contrast to state-of-the-art approaches, no additional CPU utilization occurs. The congestion control between NIC and FPGA, which causes descriptor writebacks after successfully sending packets, and tailpointer batching are disabled.

In Figure 7, four measurement results are presented. First, baseline scenario 1 in green is shown. It is apparent that the forwarding latency is mostly below $25\mu s$ with medium jitter but increases every $2s$ strongly. This pattern is very reproducible. We assume this to be caused by the DRAM controller of the CPU used for storing the packets. In general, DRAM memory technology is known to provide non-deterministic behavior due to refresh cycles. However, we could not identify the source of this jitter.

Second, the black curve depicts the latency over time for evaluation scenario 3, where the FPGA receives and sends packets over PCIe. The average latency is $7.99\mu s$ and the standard deviation $99.02ns$. The blue and red lines show the latency for evaluation scenario 4 with only receiving or sending packets via PCIe. For receiving packets via PCIe and sending them out over an Ethernet port on the FPGA we measured an average latency of $3.37\mu s$ and $40.01ns$ standard deviation. Receiving packets with the FPGA and send them out via PCIe has $4.56\mu s$ latency and $46.96ns$ standard deviation.

Note that the sum of the rx and tx latency is $60ns$ lower than the bidirectional latency. In addition, the native Ethernet ports, including the intellectual property core of the FPGA, also cause a slight latency increase. This latency variation is caused by the higher utilization of the PCIe bus as each packet is transferred twice. As shown later in Section IV-C, a higher bus utilization slightly increases the latency. All in all, we can observe that the latency for sending packets via PCIe is significantly higher than for receiving packets.

	avg. goodput	avg. latency	loss
Intel Xeon 4110	$7.77Gbit/s$	$36.63\mu s$	1.84%
AMD Epyc 7402	$9.28Gbit/s$	$13.89\mu s$	0.00%
Broadcom PEX8747	$9.29Gbit/s$	$13.12\mu s$	0.00%

TABLE I
PERFORMANCE CHARACTERISTICS OF THE RECEIVING + SENDING EVALUATION SCENARIO NR. 3. THE TRAFFIC LOAD GENERATION WAS PERFORMED WITH TCP AND A PACKET MTU LIMIT OF 1514 BYTES.

C. PCIe Infrastructure Implications

In order to investigate further the impact of PCIe architectures, we performed several input-rate sweeps on evaluation scenario 3 for different architectures. In addition, we did the same measurement for the pure software forwarding baseline implemented in DPDK (scenario 1) and the state-of-the-art approach for copying data from the network into FPGAs and back (scenario 2). For the latter, we build a high-performance DPDK application in software that is receiving all packets and handing over a pointer to the FPGA. After that, the FPGA fetches the packet via DMA and writes it back to the main memory without any processing. From there, the software applications hands the packet over to the NIC for sending. Note that this application is single-threaded and with two threads for receiving and sending the packets, the performance might be doubled. We investigated an Intel Xeon Silver 4110, AMD Epyc 7402 and a Broadcom PEX8747 external PCIe switch as shown in Figure 8. Each curve consists of at least 20 measurements, dependent on its characteristically points. All tests are performed for 10 seconds and are statistically significant as each run contains multiple millions of packets.

The baseline of forwarding packets with DPDK shows up a slight increase of latency for higher loads. Only when reaching the link speed, the average latency increases strongly. Note that this scenario is not comparable to our approach, as it does not contain the forwarding of packets to and from the hardware accelerator within the system.

The second baseline, *dpdk+fpga*, represents the state of the art. We observed an increased latency at $2Gbit/s$ rate and between $2.35Gbit/s$ and $2.4Gbit/s$ first packet loss occurs. After that point, the latency is constantly high as the system is overloaded and the packet loss increases with the input rate.

In the case of both PCIe devices being connected to the Intel CPU, we observed a latency between $6.3\mu s$ and $8.2\mu s$ in the range between $100Mbit/s$ and $7Gbit/s$. After that, the latency strongly increases and we observed the first packet loss at $8.42Gbit/s$. This means, within a range of around $1.4Gbit/s$, the increasing latency is an excellent indicator for reaching the system performance without any packet loss. We observed only packets of up to $64bytes$ arriving from the NIC on the PCIe transaction layer. In the case of the PCIe switch they were much bigger. This might be the cause of the comparable bad performance.

The results for the external PCIe switch do not show up any packet loss and are pretty good in general. Only the latency increases slightly with the input rate.

The root complex of the AMD CPU shows up surprising behavior. First, it is noteworthy that no packet loss was observed at any input rate. Second, for low rates the latency is much higher. We observed this behavior for the first packets of high throughput tests as well but could not determine the reason behind this. However, the behavior resembles the characteristics of caches. The lowest latency was observed at a rate of $5.5Gbit/s$.

In addition to the UDP test with small packets of $300bytes$, we did a TCP test with three flows and maximum packet size as shown in Table I. As TCP detects packet loss and reduces the sending rate, a packet loss of only 1.84% was detected for the Intel CPU. The average goodput is measured by TCP and does not contain the packet header overhead. $9.29Gbit/s$ TCP throughput corresponds to $9.99Gbit/s$ on the link layer.

Overall, the results for the external PCIe switch are best. This is not surprising, as these devices are made for this purpose and CPUs are currently designed for PCIe devices that want to access the system memory. However, the presented results are still all good and could be improved even better if CPU vendors would optimize for this use case.

D. Controlled vs. Uncontrolled Packet Loss

Exceeding the maximum system performance causes unavoidable packet loss as previously shown in Figure 8. However, we can distinguish between controlled and uncontrolled packet loss. As mentioned before in Section III-C it is possible to enable an optional congestion control between NIC and FPGA for transmitting packets. As soon as a packet was sent, the NIC marks the tx-descriptor entry on the FPGA as sent. By that, overruns of the tx-descriptor ring can be prevented. However, it causes an additional overhead on the PCIe bus.

Figure 9 shows measurement results for evaluation scenario 3, receiving and sending packets on the NIC via PCIe. In the case of connecting the FPGA and NIC directly to the CPU root complex, we observed a quick increase in latency and out-of-order packets on the tx-port of the NIC (marked by red shading in the background) if no congestion control between NIC and FPGA is enabled. In total, $176,761$ out of $35,784,597$ forwarded packets were out of order. In the beginning, the latency increases quickly as a queue builds up in the system due to limited PCIe performance. The results with enabled congestion control show up a similar increase of latency in the beginning but no out of order packets occur. Assuming 64 ring buffer slots, a packet size of $300bytes$ and $10Gbit/s$ link speed, this would cause a latency jitter of $15\mu s$ every time the descriptor ring runs over. This can be observed in the scenario without writeback and the devices being connected via the root complex of the CPU. In the case of enabled congestion control, this latency jitter can not be observed. The NIC is causing the remaining latency of almost $400\mu s$ as the received packets can not be transferred faster to the FPGA and packets are buffered in there. It is noteworthy that the latency in case of writing back the descriptors is higher. First, this is caused by back pressure within the FPGA caused by the congestion control mechanism. Second, only a lower sending rate can be achieved due to the additional PCIe bus overhead and thus the delay of the fixed-size buffers increases.

Without congestion control we observed a packet loss rate of 9.90%, writing back the state for each sent packet increases the loss rate to 15.79%. This means, if packet reordering is less critical than a lower rate, it might be beneficial to disable the writeback congestion control.

The last plot in the figure depicts the latency over time for enabled and disabled congestion control and the devices being connected to the PCIe switch instead of a root complex. The measured latency was exactly the same. In both scenarios, the latency is constant low as the NIC can read and write packets at line rate in the FPGA memory. Consequently, in case of having a PCIe switch, the congestion control is not needed.

These results show that a descriptor writeback might be needed if the FPGA can send packets faster than the NIC can execute these requests. In the scenario of PCIe devices attached directly to the root complex this is caused by limited PCIe peer-to-peer bandwidth. Note that this might be the case as well for 1) NICs with higher link speeds, *e.g.* 40/100 Gbit/s, or 2) in case the FPGA creates new packets or increases the packet size and by that the rate within the FPGA is higher than the tx link speed of the NIC.

E. Batching tailpointer updates

The FPGA driver of the NIC must write the tailpointer for sending and receiving packets periodically into the NIC. As previously described in Section III-C, this pointer increase can be either done for each send/received packet or only after n packets in order to reduce control overhead. By that, packets

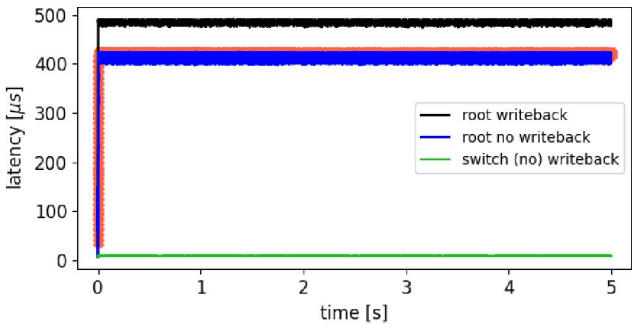

Fig. 9. Observed latency and reordered packets in case of an overloaded system with hardware accelerator and NIC directly attached to the root complex. Red shading in the background mark out of order packets. Packet reordering occured only in case of root complex and no writeback.

batch-factor/ timeout [ns]	latency [ns]		PCIe transfers [bytes/pkt]	
	average	std. deviation	outgoing	ingoing
1/-	$8.14\mu s$	$101.23ns$	324	316
8/2500	$9.43\mu s$	$98.93ns$	317	316
16/2500	$9.92\mu s$	$65.33ns$	316.7	316

TABLE II
IMPACT OF INCREASING RX/TX TAILPOINTER ONLY FOR EVERY n PACKET.

are handed over in batches to the NIC in tx direction and rx descriptors freed in batches respectively. In order to determine the exact overhead increase, we recorded and evaluated a trace of the FPGA internal PCIe data bus with the Xilinx Integrated Logic Analyzer. A tailpointer update will be only performed, *e.g.*, for every 8_{th} packet or after $2500ns$ latest. In addition, we measured the end-to-end latency and its variation for both scenarios. The results are shown in Table II.

First, an increase of the batch size lowers the written bytes per packet on the PCIe bus. The number of bytes written and read out via DMA from the NIC on the FPGA is constant, but the number of direct writes from the FPGA to the NIC has decreased. Second, it is noteworthy that the latency increases with a higher batch factor in tx direction. The NIC can not start sending a packet before being notified that there is a new packet as the NIC is not polling the descriptor ring. By that, packets are not sent out immediately after being received. In the case of packet sizes of $300bytes$, a link speed of $10Gbit/s$ and batches of 8 packets this leads to a theoretical increase of latency by $7 \cdot 300B/10Gbit/s = 1680ns$. The measured increase in latency was only $1.29\mu s$. We assume the remaining $390ns$ being contained in the baseline with no tailpointer batching but could not confirm this. In the case of 16 packets per batch, the timeout of $2500ns$ will be reached before 16 packets have been accumulated and by that, the tailpointer on the NIC will be increased every $2500ns$. Indeed, the measured latency increase is slightly smaller, presumably for the same reason. However, it is noteworthy that the latency standard deviation, which is a good indicator for jitter, has decreased. The rx tailpointer is only used for indicating free descriptor entries to the NIC and as long as enough free entries are available for receiving new incoming packets, this optimization has no negative implication on the system performance.

F. Resource Utilization

The resource utilization of the NIC driver on the FPGA is negligible in terms of logic cells and acceptable in terms of memory. It is noteworthy that a PCIe module is needed requiring some resources. Nevertheless, this module is integrated as fixed silicon by the vendor of modern FPGAs and requires only little additional programmable resources. Further, this module is also needed for hardware acceleration approaches without host bypassing. Our design for sending and receiving packets via PCIe on the Xilinx Alveo U50 utilized 2.08% of lookup tables, 1.44% of the available flip flops, 23.14% of BRAM and 0 ultra ram. In total, 311 BRAM cells were used while 128 cells are used for the rx and tx buffer each. Further 8 cells for the rx and tx descriptor rings are needed. The remaining cells are used for the PCIe module and its infrastructure, realized with intellectual property of the FPGA vendor. The utilization could be decreased by reducing the PCIe bandwidth, which is currently Gen.3 x8, and by that the input data width of the BRAM memory from $256bit$ to $64bit$. However, we did not investigate this further.

V. RELATED WORK

Reconfigurable hardware for networking purposes is an ongoing research issue. Newly introduced concepts for programmable NICs and switches enable reconfiguration with the same performance as ASICs with fixed functionality [10], but their flexibility is limited. Domain-specific languages which are made for such hardware architectures, such as P4, allow the description of the data plane behavior for use cases with low and medium complexity [11]. However, complex network functions, *e.g.*, the encoding of radio signals within the Distributed Unit of 5G O-RAN systems, can not be realized with such programmable networking hardware, and more general accelerators are needed [12], [13]. Using FPGAs as hardware accelerators for network functions not only increases the system's energy efficiency [14], we have also shown in previous work that it enables higher and more deterministic performance characteristics [15].

In addition to FPGAs, offloading network functions on GPUs has been discussed as well. GASPP is a framework for network packet processing on a GPU [16]. It uses page-locked host memory that is accessible by both an Ethernet NIC and the GPU, but must still be transferred for processing in case of discrete GPUs. Sun et al. [17] propose an extension of the Click router for packet processing on GPUs. Also their solution copies packets via the system's main memory. Kalia et al. [18] evaluate the benefits of GPUs for network packet processing critically. Their main finding is that a pure CPU-based implementation can outperform the GPU for some simple network functions due to fewer packet copies. They mention that the "NVIDIA GPUDirect" technology, which is similar to our host bypassing approach, could improve the performance. GPUnet [19] employs GPUDirect for direct communication between GPU and InfiniBand NIC using PCIe peer-to-peer, bypassing host memory. The focus of the work is, however, the simplicity of GPU programming and not network

function processing on GPUs. Moreover, these GPUDirect approaches require support in GPUs and specialized NICs that is usually not available in commodity hardware, while our solution is open source and uses standard NICs.

The benefits of direct data exchange between FPGAs and GPUs via PCIe without any CPU interaction have been shown for several applications [20], [21]. Markussen et al. [22] propose a solution for sharing devices in a PCIe-interconnected cluster system. They demonstrate similar performance benefits from peer-to-peer PCIe transfers, avoiding unnecessary copies to main memory, but they also observe a similar reduction in bandwidth as observed in our own evaluation when DMA transactions are routed through the root complex of an Intel Xeon CPU instead of a PCIe switch.

VI. Conclusion

It is crucial for many applications and use cases to have an underlying network infrastructure with in-network computing capabilities that provides high throughput, minimal latency and jitter, and avoids packet loss. This work has shown how PCIe-based hardware accelerators, specifically FGPAs, can be more efficiently integrated into the network data path, thus enabling such network infrastructures.

The presented `host bypassing` approach implements a mechanism for driving a commodity NIC from an FPGA. Our implementation allows receiving and sending network packets from commodity NICs without requiring any CPU interaction or unnecessary memory copies via system memory. Our evaluation results show that even for commodity CPU architectures, which are not optimized for direct communication between different devices, huge performance benefits can be achieved by building on peer-to-peer host bypassing.

We believe that with optimization of PCIe root complex architectures of future CPU generations for this particular scenario, even better performance can be observed.

In future work, we will investigate the host bypassing approach with GPUs. Related work has already demonstrated performance benefits from network function offloading using GPUs, which we believe can be improved further by utilizing host bypassing and relying on PCIe peer-to-peer capabilities.

Acknowledgment

This work has been funded by the Federal Ministry of Education and Research (BMBF, Germany) within the Software Campus Project "5G-PCI" and by the German Research Foundation (DFG) as part of the project C2 within the Collaborative Research Center (CRC) 1053 MAKI. We thank Xilinx and Intel for their software and hardware donations. Furthermore, we thank our reviewers for their valuable feedback.

References

[1] W. Kellerer, A. Basta, P. Babarczi, A. Blenk, M. He, M. Klugel, and A. M. Alba, "How to measure network flexibility? a proposal for evaluating softwarized networks," *IEEE Communications Magazine*, vol. 56, no. 10, pp. 186–192, 2018.

[2] Y. Li and M. Chen, "Software-defined network function virtualization: A survey," *IEEE Access*, vol. 3, pp. 2542–2553, 2015.

[3] R. Kundel, T. Meuser, T. Koppe, R. Hark, and R. Steinmetz, "User plane hardware acceleration in access networks: Experiences in offloading network functions in real 5g deployments," in *Proceedings of the 55th Hawaii International Conference on System Sciences*. Computer Society Press, 2022, p. 1–10.

[4] R. Kundel, T. Burkert, C. Griwodz, and B. Koldehofe, "Chaining of hardware accelerated virtual network functions in pcie environments," in *Proceedings of the 20th International Middleware Conference Demos and Posters*. New York, NY, USA: Association for Computing Machinery, 2019, p. 13–14.

[5] *PCI Express 3.1 Base Specification*, Peripheral Component Interconnect Special Interest Group (PCI-SIG), 2010.

[6] P. Emmerich, M. Pudelko, S. Bauer, S. Huber, T. Zwickl, and G. Carle, "User space network drivers," in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2019, pp. 1–12.

[7] T. L. Foundation, 2010, https://www.dpdk.org/.

[8] R. Kundel, F. Siegmund, J. Blendin, A. Rizk, and B. Koldehofe, "P4STA: High performance packet timestamping with programmable packet processors," in *Network Operations and Management Symposium (NOMS)*. IEEE/IFIP, 2020, pp. 1–9.

[9] B. R. Opstad, J. Markussen, I. Ahmed, A. Petlund, C. Griwodz, and P. Halvorsen, "Latency and fairness trade-off for thin streams using redundant data bundling in tcp," in *Proceedings of the 2015 IEEE 40th Conference on Local Computer Networks (LCN 2015)*, 2015, p. 287–294.

[10] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, 2013.

[11] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[12] J. Bishop, J.-M. Chareau, and F. Bonavitacola, "Implementing 5g nr features in fpga," in *2018 European Conference on Networks and Communications (EuCNC)*, 2018, pp. 373–9.

[13] J. C. Borromeo, K. Kondepu, N. Andriolli, and L. Valcarenghi, "An overview of hardware acceleration techniques for 5g functions," in *2020 22nd International Conference on Transparent Optical Networks (ICTON)*, 2020, pp. 1–4.

[14] L. Nobach, B. Rudolph, and D. Hausheer, "Benefits of conditional fpga provisioning for virtualized network functions," in *2017 International Conference on Networked Systems (NetSys)*, 2017, pp. 1–6.

[15] R. Kundel, L. Nobach, J. Blendin, W. Maas, A. Zimber, H.-J. Kolbe, G. Schyguda, V. Gurevich, R. Hark, B. Koldehofe, and R. Steinmetz, "OpenBNG: Central office network functions on programmable data plane hardware," *International Journal of Network Management*, 2021.

[16] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis, "GASPP: A gpu-accelerated stateful packet processing framework," in *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, G. Gibson and N. Zeldovich, Eds. USENIX Association, 2014, pp. 321–332.

[17] W. Sun and R. Ricci, "Fast and flexible: Parallel packet processing with gpus and click," in *Architectures for Networking and Communications Systems*, 2013, pp. 25–35.

[18] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen, "Raising the bar for using gpus in software packet processing," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 409–423.

[19] M. Silberstein, S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, and E. Witchel, "GPUnet: Networking abstractions for GPU programs," *ACM Trans. Comput. Syst.*, vol. 34, no. 3, pp. 1–31, sep 2016.

[20] Y. Thoma, A. Dassatti, and D. Molla, "Fpga2: An open source framework for fpga-gpu pcie communication," in *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2013, pp. 1–6.

[21] R. Bittner, E. Ruf, and A. Forin, "Direct gpu/fpga communication via pci express," *Cluster Computing*, vol. 17, no. 2, pp. 339–348, 2014.

[22] J. Markussen, L. B. Kristiansen, R. J. Borgli, H. K. Stensland, F. Seifert, M. Riegler, C. Griwodz, and P. Halvorsen, "Flexible device compositions and dynamic resource sharing in pcie interconnected clusters using device lending," *Cluster Computing*, vol. 23, pp. 1211–1234, June 2020.