VirtualStack: Green High Performance Network Protocol Processing Leveraging FPGAs

Jens Heuschkel
TK / TU Darmstadt
heuschkel@tk.tu-darmstadt.de

Philipp Thomasberger
TK / TU Darmstadt
thomasberger@tk.tu-darmstadt.de

Julien Gedeon
TK / TU Darmstadt
gedeon@tk.tu-darmstadt.de

Max Mühlhäuser
TK / TU Darmstadt
max@tk.tu-darmstadt.de

***Abstract*—In times of cloud services and IoT, network protocol processing is a big part of the CPU utilization today. Foong et al. proposed the rule of thumb for TCP, that a single-core CPU needs about 1 Hz clock frequency to produce 1 bit/s worth of TCP data packets. Unfortunately, CPU speed has stagnated around 5 GHz in recent years resulting in a upper limit of 5 GBit/s throughput with single-threaded network processing. Further, CPUs featuring such high clock rates (e.g., Intel Core i7-8086K) have rated TDP around 95 W, resulting in very high power consumption for high throughput situations. Meanwhile, industry offers some hardware acceleration for TCP as part of their server network cards, to relief the server CPUs and increase the energy efficiency. However this is just a small support as state and management still needs the CPU of the host system.**

In this paper, we present an approach based on field programmable gate arrays (FPGA) to not only free up CPU cycles but provide a scaleable and energy efficient concept to fully utilize high-speed network interfaces, while maintaining the flexibility of software solutions. For our evaluation, we utilized the NetFPGA Sume, proofing to achieve the linerate of connected SFP+ ports while power consumption stays below 6 W. By leveraging network protocol virtualization, the hardware acceleration approach is not only deployable but stays flexible enough to adapt new networking paradigms quickly.

***Index Terms*—protocol virtualization, FPGA acceleration, energy efficiency.**

I. Introduction

Hardware acceleration was always the solution to overcome the drawbacks of software solutions. As a result, we have today specialized hardware modules for many tasks, starting with floating point operations, memory management, or encryption units in CPUs, going to graphics rendering co-processors such as GPGPUs on separate cards. Specialized hardware always fulfills its task way better than software solutions, with the drawback of only fulfilling the one specialized task. Hence the discussion of hardware acceleration is always based on the necessity of high performance or low energy consumption for the desired task against the additional hardware cost and the additional consumed physical space of the additional chip.

In recent years, network speed has increased more rapidly as CPU clock speed. In fact, the CPU clock speed has stagnated around 5Ghz[1] and CPU hardware development moved to increase CPU core count to achieve higher system performance. If we follow the rule of thumb by Foong et al. [3], the upper boundary for single network connections is 5 Gbit/s, since current network protocol processing architecture enforce single-threaded processing. A further consequence of this rule is the 100 % utilization of one CPU core for network packet processing, that is not available for application processing at this time. As a result of the high utilization, the CPU has a high power consumption near its TDP. Hence, we argue that we reached a point where it is worth to consider hardware acceleration for high-performance networking scenarios for three reasons: i) Free CPU cycles for (server) applications instead of utilizing them for network protocol processing. ii) Significantly higher energy efficiency as CPU based network protocol processing. iii) Easy scaling for high network throughput.

Even though hardware acceleration sounds promising to tackle future network speeds, it will contribute to the ossification problem if implemented like current TCP offloading engines [3]. Since hardware implementations (especially ASICs) are static, it would take even more time to adopt new networking paradigms than in software. Therefore, the hardware acceleration has to implement the concept of network protocol virtualization [6], to retain the flexibility of adapting new network paradigms. To provide certain flexibility within the hardware part, an FPGA is the means of choice. Since an FPGA can be programmed by domain-specific hardware description languages (e.g., VHDL), updates with novel networking concepts are possible even at runtime.

Although the performance gain and the energy savings make up for the additional hardware cost at least in professional environments, the additional space consumption may be an issue in high dense server farms. As Intel recently bought Altera[2] (2nd biggest FPGA company), we assume to see FPGA and CPU combinations on one chip soon. These combined processors would significantly decrease cost and space consumption. Optimally, they would access the same memory space, resulting in seamless hardware acceleration as we know it from APU processors [2], [5].

Therefore we answer the following three research questions:

1) Where is the sweet spot between hardware and software? Although it is possible to implement every task in hardware, it may not be the best solution when it consumes too much FPGA resources or issues communication overhead between the FPGA and the host system. We

[1]https://software.intel.com/en-us/blogs/2014/02/19/why-has-cpu-frequency-ceased-to-grow (accessed 11.01.19)

[2]https://newsroom.intel.com/news-releases/intel-completes-acquisition-of-altera/\#gs.UvoygCpG (accessed 11.01.19)

investigate the consequences of porting the concept of network protocol virtualization and discuss the dimensions of resource consumption.

2) What is the acceleration potential of the FPGA based network protocol virtualization system? We provide a prototype implementation of our hardware concept to evaluate the respective performance potential.
3) What is the energy saving potential while maintaining maximal network performance? We investigate the energy consumption of our prototype implementation to discuss the dimensions between CPU and FPGA power consumption.

The remainder of the paper is structured as follows: Section II presents the related work and the background of the network protocol virtualization concept *VirtualStack*. In Section III we present the FPGA architecture. Afterwards, Section IV presents and discusses our evaluation results. Before Section VI concludes the paper, we discuss limitations of hardware implementations in Section V.

II. Background and Related Work

In this section, we give an overview of necessary background information along with the related work to this topic.

In our previous work, we presented *VirtualStack* (VS) [6]. VS represents our baseline architecture for the concept of network protocol virtualization. The basic idea is to use a shim layer between applications and network protocol stacks, to allow application-independent network protocol stack management. Therefore, VS provides modules to interface to (existing) applications, manage network stack composition, defines a protocol processing pipeline, and offers a management interface to enable management through software-defined networking concepts. The architecture also offers the possibility of application layer middleboxes (ALM), which can be utilized to enrich the network flow with additional services. In this paper, we leverage our experience with the software implementation and transfer it into hardware. The main focus is on the processing pipeline and processing related management tasks, as we think these parts are essential for the majority of use-cases.

In the field of FPGA-based acceleration, researcher and industry typically implement bare protocols or Frameworks for protocol building. Sidler et al. [10] presented a TCP/IP implementation for the Xilinx VC709 platform. Their design implements all functions of TCP and is able to saturate the 10 Gbit/s linerate of their test platform. In contrast to other implementations, they optimized their core for a high number of session counts which is prominent for common use-cases such as web servers. Another example in this category is the work of Anwer et al. [1]. They present SwitchBlade, a platform for rapidly deploying custom protocols on programmable hardware. Naturally, Switchblade's design uses a processing pipeline and uses individual hardware modules. In their evaluation, they leveraged Switchblade to implement basic network protocols such as IPv4, and an OpenFlow switch.

Also other services such as network function virtualization (NFV) were explored in combination with FPGA acceleration.

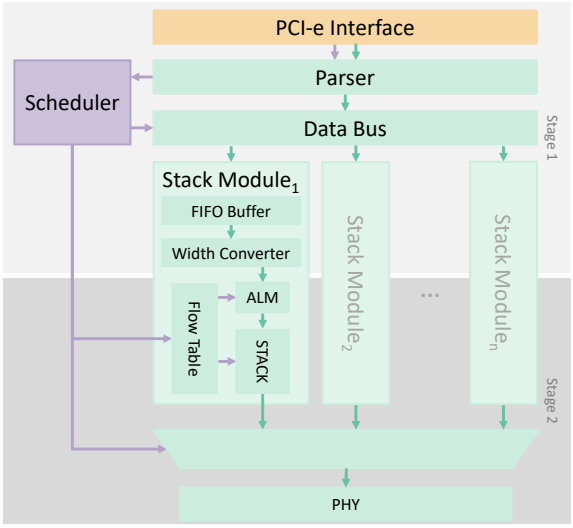


Fig. 1: Hardware architecture for FPGA-based acceleration.

Kachris et al. [7] states that NFV is mainly understood as the movement of network services from specialized hardware products towards commodity off-the-shelf hardware. They propose FPGAs as the best trade-off of both worlds, since FPGAs execute real hardware modules, and thus, deliver reliable performance with low energy consumption. Since FPGAs are re-configurable, they can be used for different services instead of fixed services as it is the case for specialized hardware products. Nobach et al. [8] identifies the cost of FPGAs as problem for NFV. As FPGAs are more expensive as standard commodity of the shelf processors, they propose dynamic provisioning as a solution. Their evaluation state a cost reduction of 39%. Both concepts leverage the dynamic reconfiguration of the FPGAs to increase cost efficiency, as we do with our stack modules.

However, direct protocol or service implementations require a modification of the corresponding software of the host system. As it is doubtful that the enormous amount of running software becomes re-implement to match the new interfaces, the solutions are not deployable in the current Internet. With our strategy of leveraging network protocol virtualization, an application modification is not required, and thus, the solution is deployable.

III. FPGA Design

As discussed above, we try to include as many VirtualStack related management tasks as possible into the FPGA design, to reduce CPU usage. We decided to include all critical processing path components for network protocol processing together with the network protocol related management (e.g., packet re-sending). Together with the application interfacing software, the network flow management remains in the software part of the concept.

Figure 1 illustrates the hardware architecture. The architecture is optimized to achieve the linerate of the physical network ports, but at the same time, it must offer the flexibility of network protocol virtualization. We also make sure that the architecture is easily scalable for higher line rates in the future.

TABLE I: Available Memory Types.

	Size on Testsystem Mbit	Access Delay # CLK
DRAM	10.89	1
BRAM	52.92	1
SRAM	216	2.5
DDR RAM	64000	28-190

Therefore, the processing is split into two stages. The first stage processes incoming data and management packets from the software. The data stream arrives in the hardware via the PCI Express interface (top of the figure). As flow management takes place on the software side, the arriving data stream contains the payload tagged with a flow id and control messages. The parser module separates payload from management data and passes management information to the scheduler. The scheduler stores management information such as the scheduling scheme and assigned network stack modules in a local look-up-table. For each data packet, the scheduler decides which stack module receives the data. Every stack module contains a FIFO buffer to store a scheduled packet. After this FIFO buffer, the second stage begins. The remainder of the stack module that is part of the second stage is treated as a black box that processes the payload into packets according to its protocol composition. Depending on the module implementation, the stack module can contain a private pipeline with additional stages, as the stack module must be able to achieve the clock rate of the second stage. To fulfill the packet processing, the network stack modules manage the addressing information required for the respective protocols for every assigned flow-id in a local look-up-table. When the packet processing is finished, the stack modules set the ready-to-send signal to indicate a sending wish to the scheduler. The scheduler decides in round-robin order which stack module is allowed to use the corresponding PHY module at a given time, whereby a PHY module represents a physical network connection.

Network stack modules are uniquely assigned to a PHY module, that is connected to a physical network port. Due to this design decision, a network stack module has to reach only the linerate of a single physical network port, which reduces the design complexity and clock rate, and thus, the energy consumption. Also this couples the second stage – and thus the processing part of the stack module – to the bus width and the clock rate of the corresponding PHY module. On the other hand, for every physical network port, a separate stack module is required, even if network stacks are identical.

The described design leads to flexible scalability for virtually arbitrary high throughput by maintaining a static latency behavior. The first stage must be configured to provide enough data to saturate the available physical network ports (in our case SFP+ modules). Two ways are possible to increase throughput: First, by increasing the bus width to transport more data within one clock cycle. Second, by increasing clock rate to transport data more often in a given time frame. The bus width is limited by the available routing lines of the underlying FPGA hardware. The clock rate is limited by the critical path of the scheduler logic and parser logic and manufacturing properties of the underlying FPGA hardware. The second stage is dominated by the available PHY modules, which themselves are dependent on the physical connection. Hence, the connection bus of the second stage has the bus width of the PHY module and uses the same clock rate. The resulting configuration will be able to saturate the underlying physical link in a properly designed network board.

TABLE II: Xilinx Virtex-7 XC7V690T Hardware Specification

Resource	Available	Used
Flip-Flop	866400	189841 (21.9%)
Look-Up-Table	693120	157504 (22.7%)
BRAM (kbit)	52920	42336 (80.0%)

Since the whole design is realized in specialized hardware and does not use a general-purpose processing module to fulfill protocols processing, the number of stack modules are limited too. The possible number of different stack modules heavily depends on the implemented protocol stack, and the used FPGA model. Limiting factors are the available logic cells and memory. Most FPGA products feature internal fast BRAM (accessible within one clock) and distributed random-access memory build from logic cells. Additionally, it is possible to attach external memory modules, such as SRAM (accessible within 2-3 clocks) or DRAM (accessible within 28-190 clocks) but due to the slower access times, sustainable throughput at linerate is hard to achieve. Hence, we only use BRAM within the critical processing path and use external memory for side tasks, such as packet resending. Table I summarizes the described properties. To partially overcome the problem of limited stack modules, the architecture is designed to support partial reconfiguration[3]. This allows changing parts of the FPGA configuration without interrupting the operation of remaining modules. Hence an inactive stack module can get reconfigured at runtime, to support other network stack compositions.

IV. Evaluation

To evaluate our concept, we implemented the described architecture. As hardware platform we used the NetFPGA SUME[4], which is a specialized FPGA developer board for network purposes. From the rich feature-set of the board we utilize the Xilinx Virtex-7 XC7V690T (see Table II), the four SFP+ interfaces (10Gbps each), and the connected SRAM (3x72Mbit). Since we needed a synchronized clock for our experiments, we decided to use the same NetFPGA board for sending and receiving concurrently. Hence, we utilized two of the four SFP+ ports as sender ports and the remaining two SFP+ ports as receiver ports. The ports are directly connected through optical fiber cables. To generate payload we flash a packet generator onto the FPGA. Figure 2 illustrates the experiment setup for one SFP+ channel. Table II shows the

[3]Partial Reconfiguration is a feature that allows flashing partial bitstreams of an FPGA design. https://www.xilinx.com/products/designtools/vivado/implementation/partial-reconfiguration.html, (accessed 11.01.19).

[4]https://netfpga.org/site/#/systems/1netfpga-sume/details (accessed 11.01.19)

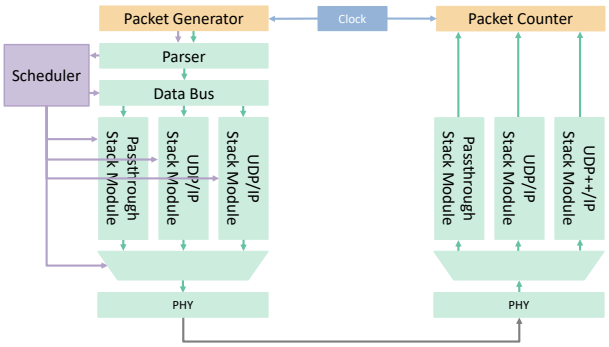


Fig. 2: Illustration of experiment setup for the evaluation.

consumed hardware resources for the experiment setup with stack specific flow tables for up to 2^{16} flows[5].

In the following, we will present the evaluation in two parts: First, we show the network performance evaluation to prove the feasibility of sufficient network performance with the presented concept. Second, we present an energy consumption discussion to demonstrate the superiority of the presented discrete hardware concept over CPU based solutions for network protocol processing.

A. Network Performance

To evaluate network processing performance of our prototype implementation, we investigate the protocol processing latency, jitter, network throughput in a logic level simulation. To confirm the simulation results, we measuread processing latency, application layer latency, and throughput using the NetFPGA with the experiment setup illustrated in Figure 2.

Figure 3(a) shows the processing latency for one network packet of a mouse flow (8 bytes payload) in clock cycles. As we used a clock rate of 156.25 MHz, a clock cycle equals to:

$$t_{latency} = n_{cycles} * \frac{1}{f_{clk}} = n_{cycles} * 6,4 * 10^{-9}$$

The prototype implementation handles 128 bit (16 bytes) per clock cycle in stage one. For every incoming network packet, the first processing stage is identical. In our implementation this takes six clock cycles (38.4 ns); two for parsing the packet, three for traversing the FIFO buffer, and one to convert the bus width to the second stage. Afterward, the payload gets processed according to the respective protocol stack. We measured three stack modules to provide an overview for overall processing latency: First, we measured a passthrough (P/T) stack for preprocessed payload, that only adds the Ethernet frame. This corresponds to the behavior of standard network cards. The processing path adds up to 13 clock cycles (82.3 ns). Second, we measured a UDP stack, processing the payload to sendable UDP/IPv4 packets including the Ethernet frame. A packet was processed within 20 clock cycles (128 ns). Lastly we added an ALM to the UDP stack to provide NACK based reliability, FEC, and in-order-delivery. This feature-set was formerly presented as UDP++ in [6], [4]. The additional layer took three additional clock cycles, and thus, a UDP++ packet was processed within 23 clock cycles (147.7 ns).

The same measurements are shown in Figure 3(b) for an elephant flow with a payload size of 1456 bytes. Due to the bigger packet size, the packet needs more time to get buffered into the FIFO buffer. Together with the parser and the width converter, this results in 95 clock cycles in the first stage. The passthrough stack has a task independent from the packet size, and thus, needs the same amount of cycles as for the mouse flow. This results in 102 clock cycles (652.8 ns). As the UDP header contains the length of the packet as well as a checksum, the whole packet needs to be buffered in order to obtain the required information for the header. The UDP module takes 192 clock cycles to process the packet because the bus width is only 64 bit inside the module. Together with the stage one processing, a UDP packet takes 294 clock cycles (1881.6 ns) to be processed. The UDP++ ALM operates independently from packet size and adds the same three clock cycles as for the mouse flow. Therefore, a UDP++ packet is processed within 297 clock cycles (1900.8 ns).

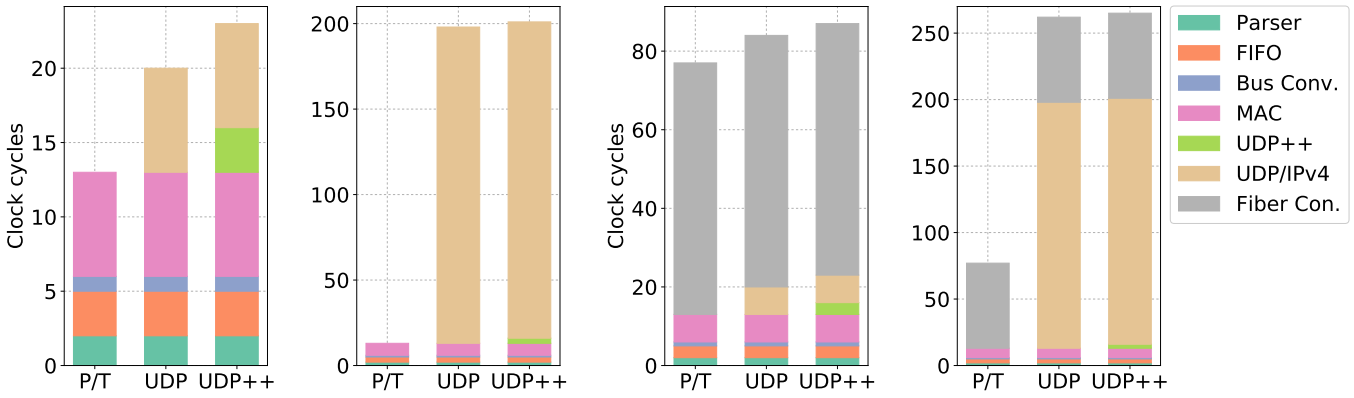
To confirm the results from the logic level simulation we carried out a real-world measurement. We performed the same experiment described above but flashed it on the NETFPGA SUME. To measure the processing latency, we added a module to count the clocks for every processing step. Fortunately, the results line up with the logic level simulation as shown in Figure 3(c) and Figure 3(d). As the jitter introduced by the SFP+ module is in the picosecond region, we were not able to measure any variances in the physical propagation delay.

From the perspective of raw network packet transmission, jitter does not occur within the processing pipeline. However, many network protocols produce management packets such as ACK, NACK, or FEC packets. The stack module must send the management packets in between the data packets. This results in a small application layer jitter since the payload processing is postponed until management packets are processed.

The architecture is designed to saturate the linerate of the existing SFP+ ports. As we explained above the stage one bus needs to fit the capacity of all physical network interfaces. Therefore we set the bus width to 128 bit and used the same clock rate as for stage two. Hence the bus is able to transmit enough data for two saturated network connections as we needed it for our experiments setup. The stage two bus is defined by the hardware connections to 64 bit and 156.25 MHz. Hence the raw throughput always reaches the full linerate of 10 Gbit/s per port. The application layer payload throughput (goodput) depends on the used protocol and the used packet size, comparable to software implementations. In contrast to software solutions, the performance is not limited by an upper packet per second boundary, but with the upper limit of raw throughput, the buses are capable to handle.

To conclude the network performance discussion, we state that the performance of the prototype implementation is outstanding in all disciplines. The measured overhead is not only negligible in networking dimensions but also several

[5]Business routers such as the Cisco RV3401W allow up to 40000 concurrent session. https://www.cisco.com/c/en/us/products/collateral/routers/small-business-rv-seriesrouters/datasheet-c78-739257.html, (accessed 27.12.18). 16 bit addresses are needed to fit these number of flows.



(a) Mouse Flow (SIM) (b) Elephant Flow (SIM) (c) Mouse Flow (HW) (d) Elephant Flow (HW)

Fig. 3: Processing latency for a mouse flow and a elephant flow in logic level simulation and hardware testbed. X-axis indicating used protocol stack whereby P/T stands for the passthrough stack.

orders of magnitude faster than the CPU-based results in our previous work, and thus, will not affect application layer network performance. As the implementation of a TCP stack module would not provide additional scientific insights for our concept, we left it for future work. However, Sidler et al. [10] investigated their TCP hardware implementation and found comparable evaluation results to ours using the UDP++ stack.

B. Energy Consumption

In this section, we discuss the power consumption of the prototype implementation of the presented design. Table III shows the power consumption of the individual modules calculated by the Xilinx Vivado Power Report. Besides our modules, the power report contains the PCI-Express modules that are necessary to drive our design from a host system and the PCSPMA modules which represent the connection to the SFP+ ports. All shown values are dynamic values, which means that this power is needed while changing its state, and thus, is proportional to the clock rate and can be avoided by setting the clock to 0 Hz. The static consumption of the design is *0.562 W* and is continuously consumed. Because most of the power draw is dynamic, the power consumption could further be reduced using clock gating. With this technique, modules that are not actively processing get cut off from the clock source, eliminating the dynamic power draw of a module during its inactive time.

TABLE III: Dynamic Power Consumption of the Individual Modules (Vivado Power Report)

Module	Power Consumption
Xilinx PCIE	2.674 W
PCIE DMA Engine	0.851 W
PCSPMA	0.428 W
UDP Stack	0.227 W
UDP++ Stack	0.228 W
Scheduler	0.080 W
Packet Generator	0.003 W
Parser	0.002 W
Databus	0.002 W

For a typical server network interface card with two SFP+ ports, only the network protocol stacks need to be doubled, and the bus size needs to be increased. Together with the static power consumption, the total power consumption of $\approx$ 6 W is added up in Table IV. If we were utilizing all four SFP+ ports, the power consumption results in $\approx$ 7.7 W. To reduce the standby power draw, some parts of the card can be deactivated through clock gating. This includes the scheduler, the network protocol stacks, and the SFP+ slots. The parser and the PCIE interface need to stay active to wake up the card if needed. Together with the static power draw, this would result in a standby power consumption of $\approx$ 4 W.

Similar to the results of our network performance measurements, the energy consumption results are convincing. If we compare the energy consumption to the example CPU from the abstract (Intel Core i7-8086K – 95 W TDP – 4 of 6 cores used for 20 Gbit/s) with the proposed rule of thumb, the FPGA design consumes around 10% of the energy the CPU would need for the same processing.

V. Hardware Limitations

In principle, it is possible to implement the complete VirtualStack concept as hardware design. However dynamic concepts like the SDN interface and rule execution would consume many FPGA resources even though they are not performance critical. Due to cost reasons, only an implementation of the critical path with direct dependent management is desirable. As we figured for our design, the available fast memory is the most limiting factor for network packet processing. Especially for reliability mechanisms where packet caching is important, the performance of the implementations heavily depends on free SRAM space, since the only alternative is DRAM, which is several orders of magnitude slower.

As we discussed above, our design was built with scaling in mind. We discussed the extension of the clock rates and the bus width as scaling factors. However, these factors also have hardware related limitations. The presented design can run with a frequency up to 383 MHz on our test hardware. A 64-bit bus can serve 20 Gbit/s network connections with a

TABLE IV: Power Consumption for our Example NIC

Module	Power / Instance	Instances
Xilinx PCIE	2.674 W	1
PCIE DMA Engine	0.851 W	1
PCSPMA	0.428 W	2
UDP Stack	0.227 W	2
UDP++ Stack	0.228 W	2
Scheduler	0.08 W	1
Parser	0.002 W	1
Databus	0.002 W	1
Static Power Draw	0.562 W	
Sum	**5.937 W**	

clock rate of 312.50 MHz. As this clock rate depends on the compiler and synthesizer tools and the underlying hardware this might be different for other FPGA products or future compilers. The bus width, on the other hand, is much less limited. Used building blocks provided by Xilinx provide bus interfaces up to 4096 bit. Together with the 312.50 MHz clock rate, the current design would be limited to 1.28 Tbit/s worth of network traffic. However, it would be possible to use multiple building blocks in parallel, but other factors like the multiplication of the needed network protocol stacks would limit the design in beforehand as the FPGA offers a limited number of logic cells.

Another limiting factor for internal throughput is the alignment of the data. The payload size is vital to maximize internal throughput. In our prototype implementation, we work with a 128-bit Bus in the first stage and a 64-bit bus as connection to the SFP+ ports. The packet payload is 1458 bytes to maximize throughput of a UDP stack with an MTU of 1500 bytes. However, to maximize the throughput through the internal buses, the payload should be dividable into 128-bit chunks without a remainder to saturate the bus communication. Since the 1458 bytes payload is not dividable by 128 bit, every 92nd packet contains only 16 bit, which is an underwhelming bus utilization of 12.5% for this single clock($\approx$ 99.0% AVG). To fix this issue, the payload size should be reduced to 1456 bytes since it would saturate the stage one bus communication. The same issue occurs in stage two: The UDP/IP protocol stack adds the 42 bytes of header resulting in a 1498 bytes packet which is not dividable by 64-bit, and thus, cannot saturate the stage two bus to the SFP+ connection. However, the bus utilization for this single clock is with 25% higher than in stage one ($\approx$ 99.6% AVG).

VI. Conclusion

In this paper we presented FPGA based hardware acceleration for VirtualStack. We presented a comprehensive design to accelerate the critical path processing including the whole protocol stack and stack specific application layer middleboxes. Further, we presented a prototype implementation to evaluate the expectable performance and energy consumption.

The evaluation of the prototype implementation showed that throughput always achieves line rate when bus alignment is respected. Goodput, however, depends on the used protocol stack and the packet size, as it is the case for software implementations. The latency of critical path processing was always < 2 μs. Considering that standard NICs typically produce a delay of 10 μs without network protocol processing [9], these values are not only negligible but spectacularly low. The same holds for power consumption: An example two-port network card was evaluated to consume less than 6 Watt when processing network protocols at line rate. Considering that such high throughput creates high CPU utilization (if achievable at all), the power savings are remarkable.

To conclude this paper, we state that FPGA based hardware acceleration is beneficial in professional server environments since it not only reduces CPU utilization but increases performance for a fraction of the energy cost. If we see cheaper FPGA devices in future (e.g., integrated in CPUs) it will become attractive for consumer devices too. Since the presented design intentionally relies on FPGAs instead of ASICs, it provides the needed flexibility to adopt new networking paradigms quickly.

Acknowledgment

This work has been funded by the German Research Foundation (DFG) as part of the project B2 within the Collaborative Research Center (CRC) 1053 – MAKI.

References

[1] M. B. Anwer, M. Motiwala, M. b. Tariq, and N. Feamster. Switchblade: a platform for rapid deployment of network protocols on programmable hardware. *ACM SIGCOMM Computer Communication Review*, 41(4):183–194, 2011.

[2] A. Branover, D. Foley, and M. Steinman. Amd fusion apu: Llano. *Ieee Micro*, 32(2):28–37, 2012.

[3] A. P. Foong, T. R. Huff, H. H. Hum, J. R. Patwardhan, and G. J. Regnier. Tcp performance re-visited. In *Performance Analysis of Systems and Software, 2003. ISPASS. 2003 IEEE International Symposium on*, pages 70–79. IEEE, 2003.

[4] J. Heuschkel, E. Fleckstein, M. Ofenloch, and M. Mühlhäuser. Udp++: Cross-layer optimizable transport protocol for managed wireless networks. In *2019 IEEE Global Communications Conference: Mobile and Wireless Networks (Globecom2019 MWN)*, pages 1–6, Waikoloa, USA, Dezember 2019.

[5] J. Heuschkel, R. Vogel, M. Blcher, and M. Mühlhäuser. Blow up the cpu chains! opencl-assisted network protocols. In *Proceedings of Local Computer Networks (LCN), 2018 IEEE 43nd Conference on*, pages 657–665, 2018.

[6] J. Heuschkel, L. Wang, E. Fleckstein, M. Ofenloch, M. Blöcher, J. Crowcroft, and M. Mühlhäuser. Virtualstack: Flexible cross-layer optimization via network protocol virtualization. In *2018 IEEE 43rd Conference on Local Computer Networks (LCN)*, pages 519–526. IEEE, 2018.

[7] C. Kachris, G. Sirakoulis, and D. Soudris. Network function virtualization based on fpgas: A framework for all-programmable network devices. *arXiv preprint arXiv:1406.0309*, 2014.

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

[9] P. Shivam, P. Wyckoff, and D. Panda. Emp: zero-copy os-bypass nic-driven gigabit ethernet message passing. In *SC'01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, pages 49–49. IEEE, 2001.

[10] D. Sidler, G. Alonso, M. Blott, K. Karras, K. Vissers, and R. Carley. Scalable 10gbps tcp/ip stack architecture for reconfigurable hardware. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 36–43. IEEE, 2015.