Demystifying the Performance of XDP BPF

Oliver Hohlfeld
Brandenburg University of Technology, Germany
oliver.hohlfeld@b-tu.de

Johannes Krude, Jens Helge Reelfs, Jan Rüth, Klaus Wehrle
RWTH Aachen University, Germany
{*krude,reelfs,rueth,wehrle*}@comsys.rwth-aachen.de

***Abstract*—High packet rates at $\geq$ 10 GBit/s challenge the packet processing performance of network stacks. A common solution is to offload (parts of) the user-space packet processing to other execution environments, e.g., into the device driver (kernel-space), the NIC or even from virtual machines into the host operating system (OS), or any combination of those. While common wisdom states that offloading optimizes performance, neither benefits nor negative effects are comprehensively studied.**

In this paper, we aim to shed light on the benefits *and* shortcomings of eBPF/XDP-based offloading from the user-space to *i)* the kernel-space or *ii)* a smart NIC—including VM virtualization. We show that offloading can indeed optimize packet processing, but only if the task is small and optimized for the target environment. Otherwise, offloading can even lead to detrimental performance.

I. Introduction

Increasing network speeds challenge the packet processing performance of current network stacks. A common solution to this challenge is to bypass the kernel (e.g., [1], [2]) and process packets entirely in user-space, alternatively, packet processing can be *offloaded* into the kernel-space (e.g., [3], [4]) or even to dedicated hardware (e.g., [5], [6]). Typical offloading targets include in-kernel virtual machines (e.g., eBPF/XDP [4]) or specialized hardware (e.g., SmartNICs [7]). These approaches promise improved packet processing performance by shortcutting the data path given that fewer software layers need to be traversed, leading to higher throughput or shorter delays. While offloading can be perceived as a general solution to speed up packet processing, it only yields performance improvements if the offloaded tasks are actually executed faster in the target environment. As we will show, this is not always the case.

A second practical challenge is that offloading typically requires to utilize target specific APIs or frameworks. This hinders to seamlessly move packet processing tasks between offloading targets or requires to adapt to the available targets. With the introduction of the extended Berkeley Packet Filter (eBPF) to the mainline Linux kernel, this situation has changed. BPF provides an independent programming framework to define packet processing tasks that can be executed in different target environments with native performance. To this end, the eXpress Data Path (XDP) [4] provides a safe execution environment to run eBPF code in kernel-space within the device driver context. Further, SmartNICs such as Netronome's Agilio platform [7] enable the execution of eBPF on the NIC itself. Thus, eBPF provides a framework to enable offloading for a broad set of applications.

978-1-5386-9376-6/19/$31.00 ©2019 IEEE

In this paper, we empirically assess the performance of eBPF-based offloading of packet processing tasks to provide a first intuition when offloading is beneficial and when it is not. We base this assessment on evaluating three common offloading environments: the user-space via AF_XDP-based kernel-bypass, in-kernel virtual machines via XDP, and a network interface card. We then consider two use cases: *i)* offloading from non-virtualized user-space programs and *ii)* user-space programs running within VMs. The latter addresses the common case of packet processing tasks executed in cloud environments with even more performance impairing software layers introduced by virtualization. As generally expected, our preliminary results show that offloading indeed often yields performance improvements. However, as we will show, they vary by task and offloading can even lead to detrimental performance figures when offloading tasks that cannot be efficiently executed. This highlights that offloading is not always an adequate solution to improve packet processing performance. We posit that a deeper understanding of offloading effects is necessary, especially with the recent advent of offloading for the masses by eBPF.

Structure. Section II discusses related works before we explain our measurement setup in Section III. Section IV continues by analyzing offloading performance in non-virtualized settings before Section V shows the impact when virtualization is added. Finally, Section VI concludes the paper.

II. Related Work

Research identified bottlenecks in packet processing long ago and has proposed solutions by either moving processing down the network stack (e.g., [8]) or by offloading *some* processing into deeper layers (e.g., [9])—a well explored concept. Researchers pushed the limits by building [6] or using programmable hardware [10]. Utilizing the GPU for multi-pipeline and -processing lead to further hardware accelerated solutions, e.g., [5]. Hardware dependent optimizations are complemented by software designs, e.g., by splitting the control- and data plane in the OS [11], leverage microkernels [12], [13], or bypass the generic OS stack [1], [2].

However, improved networking performance does not necessarily rely on such drastic measures. In [3], the authors show that offloading of *some* processing into the OS network stack achieves reasonable improvements. Today, such measures are available off-the-shelf with the introduction of eBPF and the XDP kernel interfaces [4]. Further, their rising popularity has paved them their way into programmable hardware, e.g., Netronome SmartNICs [7] support eBPF/XDP offloading. While [4], [14] show *some* performance measurements, they do

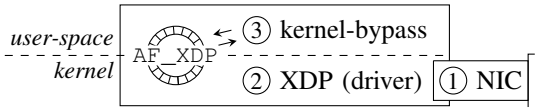


Figure 1: The execution points for the offloaded programs.

not provide a comprehensive and comparative view on different offloading setups. We complement these works by shedding light on pitfalls and possibilities of various setups that offload processing to hardware or XDP in contrast to kernel-bypass and extend this view to multi-tenant virtualized setups.

III. Experimental Setup

We compare the packet processing performance at three different points which are illustrated in Figure 1. ① We process packets on a Netronome Agilio CX 2x10GbE SmartNIC by putting an XDP program onto the NIC. Dropping or responding to packets on the NIC itself should ideally yield the highest performance since packet processing is limited by neither the main CPU nor the OS. However, this offloading is bound to the availability of a SmartNIC. ② As a more generally available option, we offload a program into the NIC's device driver. Handling packets in the device driver omits any overhead of the generic OS networking stack. Therefore, the limiting factor is mainly the Intel Core i7-7700 CPU, and the Netronome device driver accepting packets from the NIC and calling the offloaded XDP program for each packet. ③ For comparison, we also execute XDP programs compiled for user-space. We therefore use the Linux `AF_XDP` socket interface which provides high throughput by utilizing stack-bypassing (alike DPDK/netmap). Yet, we use the generic `AF_XDP` mode eliminating most, but not all of the generic packet processing of the Linux 4.18.10 kernel (in contrast to an optimized device specific mode the used NIC's driver does not support).

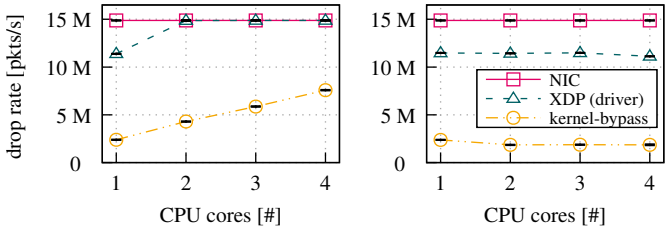
In all measurements, we compare the execution of the same XDP program at these different execution points. Traffic is generated by up to four other machines connected to the Netronome NIC through a Netgear XS728T 10 GBit switch. We measure packet rates, update rates, and CPU usage by tracking counters over 30 s intervals and repeat each individual measurements 10 times.

IV. General Offloading Performance

We begin by evaluating the traditional use case of offloading packet processing to the *i)* NIC, *ii)* kernel-space, and *iii)* via kernel-bypass to user-space. By showing the offloading effect on throughput and latency subject to the programs' CPU and memory complexity, we highlight achievable performance gains and situations in which offloading is detrimental.

A. DoS Mitigation: Dropping at Line Rate

As a performance baseline of our testbed, we evaluate the number of packet drops, i.e., as required for DoS protection as a common use case. That is, we estimate the upper bound of achievable packet rates by executing an XDP program which immediately drops all packets and increments a drop counter. To capture scalability and influence of the traffic distribution,



(a) Multiple Flows (b) Single Flow

Figure 2: Maximum packet rate when dropping all packets.

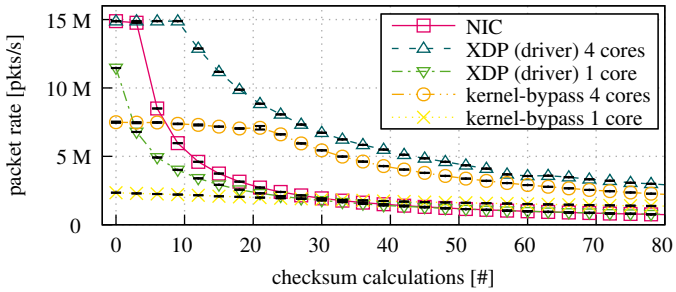


Figure 3: The achievable packet rates when varying the amount of processing for each packet.

we vary the number of used CPU cores and test minimum sized packets with *i)* multiple flows and *ii)* only identical packets.

We show the mean measured drop rate as color-coded symbols and the 99% confidence intervals with black bars in Figure 2. In line with related work, our results highlight large differences in achievable packet rates at the different execution points; while our kernel-bypass never achieves line-rate, both offloading variants can. That is, the program offloaded to the NIC always achieves the full 10 Gbit/s drop rate of 14.88M pkts/s. By utilizing RSS, device driver (XDP) offload performance scales with the number of CPUs if the traffic contains multiple flows (cf. Figure 2a). Device driver offloading suffices to achieve line-rate on our multi-core system. In the case of single flow traffic (cf. Figure 2b), usual RSS does not distribute flow processing to multiple CPU cores therefore not improving packet processing performance on multiple cores. Thus, when deciding how to offload, different traffic distributions and their effects must be taken into account.

Takeaway. *As generally known, offloading packet processing improves the achievable packet rate. This rate, however, depends on where and how the offloaded program is executed. Offloading to the device driver can already perform at line-rate, but only if the workload can be processed on multiple cores.*

B. Influence of Processing Complexity: When the NIC is Slow

The packet processing performance is not only influenced by the decision on where to offload but also by the amount of required processing. To study this effect on a generic processing workload, we next perform a varying number of computations on each packet before dropping them.

We show the achievable packet rates for all execution points when performing a varying number of IP checksum computations as a proxy measure for processing complexity in

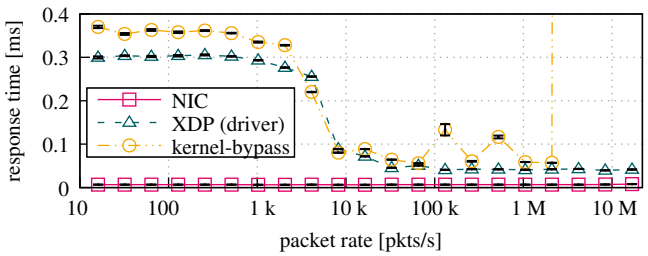


Figure 4: Response times to DNS ping.

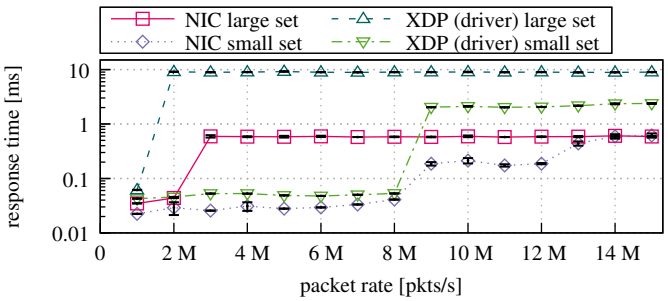


Figure 5: Delay when looking up DNS responses in memory.

Figure 3 which of course does not generalize to every kind of processing. The results highlight a performance decrease with an increasing processing complexity. Interestingly, the packet rate drops much faster when offloading to the NIC's CPU in comparison to all other cases. The NIC already performs worse than the device driver (4 cores) when only doing 6 checksums per packet due to a slower CPU on the NIC than in the host system. This shows that offloading to hardware can yield detrimental performance. Additionally, with an increase in the amount of processing the difference between in-kernel and kernel-bypass (user-space) decreases, since the fixed overhead of the kernel-bypass shrinks relative to the increasing amount of packet processing. Hence, the most gain for offloading can only be achieved when the offloaded functionality involves only limited processing.

Takeaway. *The performance of offloading tasks depends on the processing capabilities of the offloading target. With our SmartNIC, offloading complex tasks from a fast in-host CPU to a slower on-NIC CPU can slow-down packet processing instead of achieving desired performance improvements.*

C. Reducing Latency with Offloaded Responses

Offloading packet processing promises to reduce delay by involving fewer software layers. To complement our throughput perspective with a view on latency, we next measure the influence of responding at different execution points at varying packet rates. The offloaded program implements a DNS ping by responding to all our identical minimum sized equally spaced DNS requests with a *hardcoded* `NXDOMAIN` answer. The response time is determined by taking hardware timestamps for requests and responses on an Intel X550-T2 NIC connected to a mirror port on our switch.

As shown in Figure 4, offloading to the NIC reduces response times. The difference between kernel-bypass and the device driver is, however, small compared to the influence of the packet rate. At low packet rates, the response time is likely dominated by the interrupt delivery and CPU wake-up time. Response times for both kernel-bypass and the device driver drop, once the CPU switches to polling mode. To achieve low on-CPU response times, the system needs to be kept busy, e.g., with higher packet rates, or busy waiting for new packets. Kernel-bypass response times drastically increase above 2M pkts/s since the CPU gets overloaded with requests.

Takeaway. *Offloading processing tasks to the device driver (XDP) can provide minor per-packet latency improvements, depending on the processed packet rate. Large improvements can be obtained by offloading to the NIC whose latency figures are independent of the packet rate with a maximum of 16 µs.*

D. Memory Lookup Influence: Delivering Cached Responses

To study the influence of memory access, we extend the DNS program to respond with different 16 Byte resource records retrieved from an in-memory hash map. We generate two different traffic classes: *i)* the small set consists of 36^2 different DNS requests resulting in a response set of 20 KiB and, *ii)* the large set contains 36^4 DNS requests resulting in 26 MiB.

The response time in Figure 5 again varies with the packet rate. In all four shown variants, the response time drastically increases once the system becomes overloaded. For the small set, both the device driver and the NIC get overloaded at the same rate of 9M pkts/s, whereas for the large set, the NIC is able to handle a slightly higher packet rate than the device driver. Once the system gets overloaded, the NIC still produces lower response times than the device driver, however both experience packet loss. The response times from the overloaded device driver differs between both sets whereas the NIC eventually yields the same response times for both sets, probably caused by the employed memory caching strategy in the Intel CPU and Netronome NIC. Although overall the NIC produces smaller response times, when accessing the memory, the NIC becomes overloaded at a similar point compared to the device driver.

Takeaway. *Memory accesses can significantly affect the performance especially when overloaded. The NIC only shows slight advantages in maintaining small response times when performing a memory access at increasing packet rates.*

E. Memory Update Influence: Updating the Response Cache

An offloaded response cache is only useful when its entries can be updated. When updating data stored on the NIC, changes not only have to cross into kernel-space but also need to be transmitted to the NIC. To study the performance of memory updates, we continuously write from user-space our small and large response set into the hash maps.

The update rates in Figure 6 show several orders of magnitude in between the execution points. Kernel-bypass and the device driver are influenced by the size of the updated set, whereas the NIC shows no difference between both sets. By taking more than 40 seconds to update the large set on the NIC (26 MiB DNS resource records), the applicability of NIC offloading of cached responses highly depends on the required update rate.

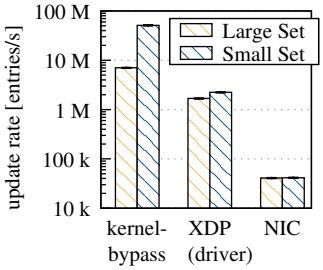


Figure 6: The rate at which the data at different execution points can be updated.

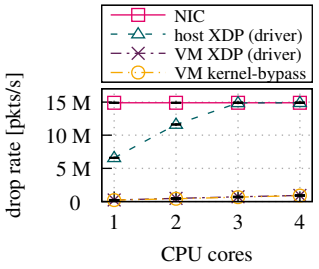


Figure 7: Maximum packet rate when dropping all packets for a VM.

Takeaway. *The choice of execution points also influences the memory update performance. Thus, depending on the use case some offloading options may not offer required update rates.*

V. Offloading from Virtual Machines

Networked applications are often executed inside VMs, especially in cloud environments. Virtualization adds additional layers as packets traverse the NIC and the host OS, e.g., via Open vSwitch, before they are forwarded over a virtual NIC to a VM. Here, we suspect that the potential for improvements through offloading are even higher. Handling packets in the host OS device driver or the NIC both removes the overhead of Open vSwitch processing and forwarding packets over the VM boundary. Since VMs are often used in multi-tenant scenarios, we also suspect that such offloading may lead to undesired influences on neighboring VMs on the same host.

A. Maximum VM Packet Rate

To assess the potential for offloading from a VM, we repeat the baseline drop measurements (cf. Section IV-A) at VM-specific execution points. Therefore, we run a VM on the Xen hypervisor (Version 4.9.2) with networking access provided through Open vSwitch running in the host. We compare packet drop rates in the VM via kernel-bypass in user-space (AF_XDP sockets) to drop rates in the VM virtual NIC driver (via XDP), dropping them in the host OS device driver (dom0 XDP) and on the NIC. Since the Xen virtual NIC driver does not yet support XDP, we use our own variant which does support XDP.

As shown in Figure 7, the achievable drop rate is significantly lower when forwarding packets into the VM. In existing virtualization solutions, offloading to the VM device driver is the only available offloading point without exposing full access to the host OS. However, as our measurement shows, the difference in achievable drop rate between VM kernel-bypass and VM device driver is relatively small with a mean drop rate of 874k pkts/s compared to 921k pkts/s when using 4 CPU cores, respectively. Although the achieved drop rate in the host device driver is smaller in comparison to our previous measurements (see Figure 2), a VM may achieve a huge gain by offloading packet processing to the host OS or to the NIC if such an offloading solution is available.

Takeaway. *Offloading within a VM is less beneficial in comparison to potential benefits of offloading packet processing from a VM to the host.*

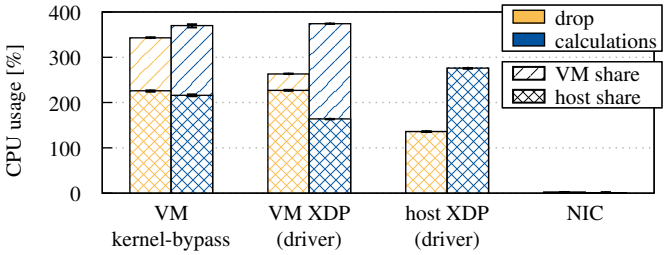


Figure 8: CPU shares for different execution points.

B. Isolation in Case of Offloading

VMs are often used as an isolation barrier between the host OS and multiple tenants sharing the same hardware. If an offloaded program is granted full access to the host OS or the physical hardware, a VM may use the offloading mechanism to break isolation. In the case of XDP, some protection is provided through the Linux kernel verifying memory safety before executing an XDP program. To allow concurrent offloading from multiple VMs, such mechanisms must be extended to also protect the offloaded programs from each other.

When executing an offloaded program on behalf of a VM within the host OS on all traffic from a NIC, this program may access, drop, and modify all traffic. Therefore, an offloading mechanism for VMs needs to limit the execution of the offloaded code by, e.g., only executing it when the destination IP address belongs to the VM in question.

To continue, we assume such isolation exists. Since offloading also shifts computation from one domain to another, we investigate the effects CPU utilization in the next section.

C. VM CPU Usage

When processing packets for a VM, the CPU performs work not only in the context of the VM itself but also in the host OS. Since CPU usage in the shared host OS is not accounted to any VM, we investigate the influence on CPU usage while offloading to different points in the virtualized stack. We execute two different programs: the drop program minimizing computations and the checksum program from Section IV-B configured to compute 150 checksums per packet before dropping it. For both, we send traffic at 800k pkts/s (below the maximum packet forwarding rate for the VM as shown in Figure 7). VM and Host OS each use 4 virtual CPU cores pinned to the 4 physical cores. We measure CPU shares of the VM and host OS via counters from the Xen hypervisor.

Figure 8 shows CPU usage of the VM and the host as a stacked bar plot. Offloading packet drops deeper into the stack lessens CPU usage by avoiding successive packet forwarding steps. Moving the program from the VM kernel-bypass to the VM device driver lessens VM CPU usage but does not affect the host OS CPU utilization since it still forwards all traffic to the VM. When dropping in the host's device driver or the NIC, host CPU usage decreases as well. Offloading packet dropping frees up CPU resources when the offloaded program does not perform any additional calculations.

This picture changes when performing calculations in the offloaded program. Since the additional calculations in the

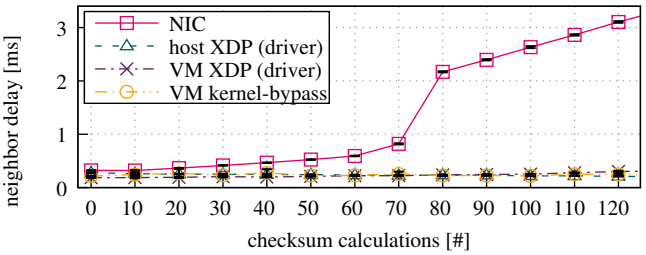


Figure 9: Delay to a neighbor when offloading computation.

host's device driver outweigh the savings from not forwarding the packets to the VM, the host's CPU usage increases due to offloading. Although much work is performed on behalf of a VM, existing accounting mechanisms would perceive the VM as not using any CPU resources, since all work is shifted to the host OS. I.e., a malicious VM may abuse offloading mechanisms to perform costly computations in the host.

Takeaway. *We conclude that an offloading mechanism needs accounting, thus, limiting the amount of offloaded computation and possibly informing the VM scheduler about offloaded CPU usage. Otherwise, offloading bears the risk of introducing unaccounted CPU usage in the shared host.*

Since the NIC is typically also shared between multiple VM, we examine the influence on a second VM in the next section.

D. Influence on Delay for Neighboring VMs

Offloading parts of one (virtualized) application may have an influence on another (virtualized) application running on the same host. In our next measurement, we look at the influence on a delay-sensitive application when offloading computation. We measure the response times for the DNS ping (cf. Section IV-C) executed via kernel-bypass in the user-space of one VM while a second VM offloads a varying amount of checksum calculations to different parts of the stack. Both VMs are connected to our switch over two different interfaces of a single NIC. We generate DNS traffic at 10k pkts/s and send 800k pkts/s (as before) to the checksum calculations.

The delay introduced for the DNS ping is shown in Figure 9 for a varying amount of offloaded checksum calculations. Performing checksum calculations has little influence on the delay of the neighbor VM as long as the calculations are performed in the VM or in the host's device driver. However, when offloading calculations to the NIC, the delay for the neighboring VM increases drastically with the amount of per packet calculations. We observe a noteworthy delay increase at 70-80 checksum calculations which is likely the point when the NIC becomes overloaded. Despite both VMs using different interfaces, the NIC apparently shares internals for both. Thus, both the DNS traffic and the checksum traffic are affected although the former is not subject to any calculations on the NIC. Performing computations on our NIC for some traffic has the potential to introduce delay to all traffic for this NIC.

Takeaway. *Offloading packet processing to the host OS brings the potential of removing the costly packet forwarding to the VM. Our used SmartNIC can introduce significant delays for all non-offloaded applications when performing offloaded computations. This must be taken into account especially for shared infrastructure.*

VI. Conclusion

In this paper, we demonstrated the potential benefits and shortcomings when using the new widely available offloading mechanisms of the Linux kernel. To this end, we investigate generic AF_XDP kernel-bypass, XDP device driver offloading and even offloading XDP programs to a Netronome SmartNIC. We further show how these approaches are challenged in the presence of virtualization. Our results indicate that offloading *can* accelerate packet processing but *only* if the task remains small. Especially our SmartNIC gets easily overloaded by too heavyweight tasks. Furthermore, updating offloaded data may be a costly operation. Yet, our SmartNIC excels when it comes to ultra-low latency processing of small tasks. Virtual machines benefit if data can be offloaded to the VM host, however, care needs to be taken to guarantee isolation and fairness. While it frees resources for other VMs, offloading further to the NIC may negatively impact the responsiveness of other VMs. Thus, in line with conventional wisdom, we conclude that Linux's general offloading framework for the masses offers great potential, however, each use case is unique and actual benefits must be evaluated individually.

Acknowledgements. This work has been funded by the DFG as part of the CRC 1053 MAKI and SPP 1914 REFLEXES.

References

[1] L. Rizzo, "Netmap: A Novel Framework for Fast Packet I/O," in *USENIX Security*, 2012.
[2] "Intel DPDK," http://dpdk.org.
[3] O. Hohlfeld, H. Reelfs, J. Rüth, F. Schmidt, T. Zimmermann, J. Hiller, and K. Wehrle, "Application-Agnostic Offloading of Datagram Processing," in *IEEE International Teletraffic Congress*, 2018.
[4] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel," in *ACM CoNEXT*, 2018.
[5] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-accelerated Software Router," in *ACM SIGCOMM*, 2010.
[6] G. Gibb, J. W. Lockwood, J. Naous, P. Hartke, and N. McKeown, "NetFPGA – An Open Platform for Teaching How to Build Gigabit-Rate Network Switches and Routers," *IEEE Transactions on Education*, 2008.
[7] J. Kicinski and N. Viljoen, "eBPF Hardware Offload to SmartNICs: cls_bpf and XDP," in *Netdev 1.2*, 2016.
[8] M. E. Fiuczynski, R. P. Martin, T. Owa, and B. N. Bershad, "SPINE: A Safe Programmable and Integrated Network Environment," in *ACM SIGOPS Workshop*, 1998.
[9] D. A. Wallach, D. R. Engler, and M. F. Kaashoek, "ASHs: Application-Specific Handlers for High-Performance Messaging," *ACM SIGCOMM CCR*, 1996.
[10] A. Fiessler, S. Hager, B. Scheuermann, and A. W. Moore, "HyPaFilter: A versatile hybrid FPGA packet filter," in *ACM/IEEE ANCS*, 2016.
[11] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A Protected Dataplane Operating System for High Throughput and Low Latency," in *USENIX OSDI*, 2014.
[12] B. N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. G. Sirer, "SPIN: An extensible microkernel for application-specific operating system services," in *ACM SIGOPS Workshop*, 1994.
[13] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, "The Click Modular Router," *ACM SIGOPS Operating Systems Review*, 1999.
[14] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, and G. Carle, "Performance Implications of Packet Filtering with Linux eBPF," in *IEEE International Teletraffic Congress*, 2018.