Performance Implications at the Intersection of AF_XDP and Programmable NICs

Marco Molè
Politecnico di Milano
Milan, Italy

Farbod Shahinfar
Politecnico di Milano
Milan, Italy

Francesco Maria Tranquillo
Politecnico di Milano
Milan, Italy

Davide Zoni
Politecnico di Milano
Milan, Italy

Aurojit Panda
New York University
New York, United States

Gianni Antichi
Politecnico di Milano
Milan, Italy

ABSTRACT

AF_XDP is emerging as an easier way to implement zero-copy network bypass applications. This is because it allows mixed-mode deployments, where zero-copy and socket-based applications share the same NIC. However, AF_XDP relies on NIC hardware and driver features, but implementing these features on programmable NICs adds resource overheads and increases development complexity and thus might not be desirable. To address this, we examine the feasibility of using eBPF based kernel extensibility to implement the required features, and report on the tradeoff between an eBPF and a native NIC implementation. Our analysis involved updating the OpenNIC driver to support the loading of eBPF/XDP programs and zero-copy AF_XDP . Our implementation is of independent interest because it makes it easier to develop and evaluate alternate designs for mixed-mode zero-copy deployments, and new NIC accelerated applications. Our implementation is open-sourced

CCS CONCEPTS

• **Networks** → **Middle boxes / network appliances**; • **Hardware** → **Hardware accelerators**; • **General and reference** → Performance;

KEYWORDS

eBPF; AF_XDP; Programmable NICs

ACM Reference Format:
Marco Molè, Farbod Shahinfar, Francesco Maria Tranquillo, Davide Zoni, Aurojit Panda, and Gianni Antichi. 2025. Performance Implications at the Intersection of AF_XDP and Programmable NICs. In *3rd Workshop on eBPF and Kernel Extensions (eBPF '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3748355.3748359

1 INTRODUCTION

AF_XDP (which has been enabled by eBPF) is emerging as a popular way to write high-performance kernel-bypass networked applications that run on Linux [10, 14, 19, 21, 30–33]. This is because unlike other kernel-bypass libraries (e.g., DPDK [8], netmap [28]), AF_XDP enables mixed-mode deployments where a NIC can be shared between kernel-bypass applications and those that use the traditional socket API (and do not bypass the kernel).

	Current Practice	Our Proposal
Advanced Flow Steering	NIC (Hardware)	+ Kernel
TX NAPI budget	NIC (Driver)	+ Kernel
NAPI instance placement	NIC (Hardware)	+ Kernel

Table 1: *List of* AF_XDP *features we consider in this paper. Currently, they are implemented either in the NIC hardware or its driver. We propose allowing programmers to use also kernel implementations.*

AF_XDP depends on NIC hardware features to provide comparable performance as other zero-copy networking libraries: it depends on hardware flow-steering (aRFS [7]) to direct all packets that an AF_XDP application will process to a single NIC queue, which the application then accesses using an AF_XDP socket (XSK). Furthermore, the send path of an XSK runs in a NAPI instance in the kernel. What core this NAPI instance is executed on can significantly impact application performance, and the NAPI instance is frequently run on a different core than the application thread. Thus, sending packets over a XSK might require scheduling a NAPI instance on a remote core, and Linux currently uses NIC interrupts for this purpose. Additionally, like other high-performance network libraries, AF_XDP depends on batching to amortize the cost of sending (and receiving packets). But batch-size can have significant impact on performance, and rather than relying on application, AF_XDP assumes that NIC drivers embed the appropriate batch size. Commodity NICs from Intel and Mellanox implement the necessary hardware features to enable zero-copy mixed-mode deployments, and embed batch-size and other necessary information in their drivers.

But FPGA-based programmable NICs are also becoming more common, and are in use in datacenters [17, 18] and other deployments [5, 9]. These NICs do not automatically come with the hardware features assumed by AF_XDP , but programmers can implement these features at the cost of added complexity and resources. Furthermore, application logic can affect the ideal batch size, and thus no single batch-size value embedded in a driver is likely to work across all deployments. Thus, one can ask *how should AF_XDP and programmable NICs evolve to better support zero-copy network applications running in mixed-mode deployments?*

Among the possible answers to the question, in this paper we focus on two (Table 1): one that matches current practice and requires that all programmable NICs implement features required by AF_XDP and that programmers update drivers when changing

eBPF '25, September 8–11, 2025, Coimbra, Portugal

ACM ISBN 979-8-4007-2084-0/25/09.
https://doi.org/10.1145/3748355.3748359

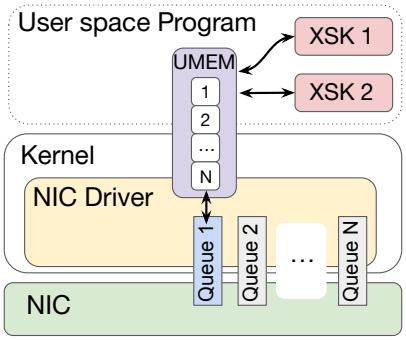


Figure 1: *XSKs explicitly bind to NIC's queues.*

NIC features; and another that exploits eBPF-enabled kernel extensibility to support NICs that do not implement all of the required AF_XDP hardware features and minimize driver changes.

In this paper, we compare these two options in the context of OpenNIC, a programmable FPGA-based NIC (§3). We evaluate the trade-offs between these options, and find that (a) while replacing hardware features (aRFS) with a kernel extension running in software has performance overheads, these overheads decrease with application complexity; (b) dynamically changing batch sizes rather than embedding it in the driver allows applications to better trade-off throughput and latency; and (c) using programmable scheduling (e.g., enabled by approaches like ghOSt [20]) for placing NAPI instances can significantly improve receive-and-transmit throughput. In sum, these results show that eBPF based kernel extensibility reduces the effort and improves the performance of AF_XDP when used with programmable NICs.

In the process of evaluating these options, we have implemented eBPF/XDP and zero-copy AF_XDP support for OpenNIC, and our code is open-source [25]. We report on the effort required, and the performance of the resulting prototype in §4. The resulting system and evaluation is of independent interest to the community, because it makes it easier (a) to develop new NIC accelerated application; and (b) to evaluate alterative designs for mixed-mode zero-copy network libraries.

2 BACKGROUND

In this section, we first provide background information on AF_XDP [22] (Section 2.1) and then introduce OpenNIC [1] (Section 2.2), the hardware/software code-base we used as reference for a state-of-the-art programmable NIC.

2.1 AF_XDP

AF_XDP is an address family in Linux for processing raw L2 frames. AF_XDP sockets (or XSKs) support zero-copy operations through use of a shared memory region between user-space and kernel called UMEM, which is comprised of fixed sized chunks used for holding packets [22]. This socket family provides an on-demand kernel bypass functionality to applications. It relies on an eBPF/XDP program (running at NIC driver) to filter and redirect traffic to the XSK (bypassing rest of the kernel) or pass it to the network stack.

A system using AF_XDP has a user-space context and kernel-space context, like any other socket family. In the user-space, the

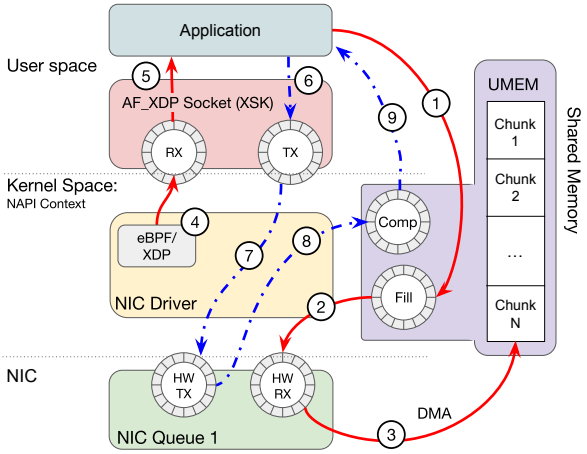


Figure 2: *Movement of a UMEM chunk during the lifecycle of an XSK. Red arrows are the receive path, blue arrows are the transmit path.*

application thread (user-space) interacts with the socket while kernel processing happen in the form of a NAPI[1] instance, an interface between a network driver and rest of the kernel. Each queue has a corresponding polling function, called NAPI [4] instance, that is triggered by the queue's interrupt line and runs until there are pending packets in the queue. Each NAPI instance runs independently of the others and generally on different cores.

Binding sockets, UMEM and queue (user and kernel context). Figure 1 shows an overview of a system using XSKs. Application (running in user-space) creates a UMEM by allocating memory and registering it with the kernel, then when creating a new XSK, application will specify a UMEM, a network interface, and a queue to which XSK will bound. The XSK will have communication channels (named Rx and Tx rings) for notifying application of arrived packets (specifying the offset in the UMEM where packet was DMAed), and to inform NIC of packets that should be transmitted. The UMEM will have controlling channel (FILL and COMPLETION rings) used for advertising available memory chunks for receive operation and notifying the application of chunks that has been transmitted. Both the XSK and the UMEM rings are Single-Producer Single-Consumer (SPSC) buffers, designed to operate without requiring read/write locks. This is good for performance, as applications can dedicate one core to the XSK (user-space) and another core to the NAPI processing (kernel-space). The details life-cycle of a packet through AF_XDP system is as follows.

Buffer allocation (user context). Figure 2 shows journey of a packet entering from hardware receive ring until when it exits the system and is placed on the transmission ring. ① In the beginning, the application is responsible for populating the FILL ring with pointers to available UMEM chunks. ② The kernel (driver) will consume the FILL ring entries, converting the pointers into a format that the hardware can interpret, called descriptors. ③ The descriptors are then sent to the NIC, which will use them as destination addresses for the DMA transfers.

Receive Path in Driver (kernel context). When a batch of DMA transfers is done, the NIC raises the interrupt line corresponding to the queue that received the packets. The interrupt service routine

[1]New API

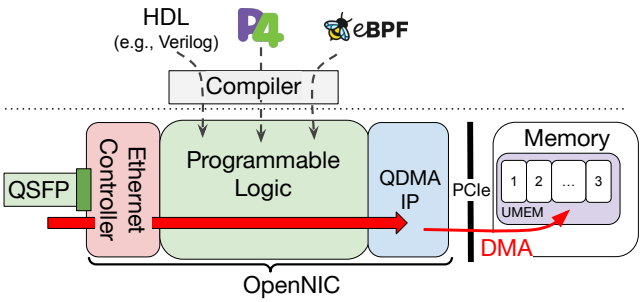


Figure 3: *OpenNIC High Level architecture*

schedules the NAPI instance responsible for polling the queue that will disable the interrupt line and start polling the HW RX queue until there are pending packets. ④ For each packet, the eBPF/XDP program is run. The program can read, modify the packet and then drop it, pass it to the network stack, or redirect it to one of the XSK bound to the hardware queue. When a packet is redirected to an XSK, its UMEM address is enqueued in XSK's RX ring.

Receive path in application (user context). ⑤ Application polls the RX ring of its XSK to receive L2 frames and process them. Here, the user-space has the ownership of the packet (UMEM chunk) and can perform any custom processing on it. It is the application's responsibility to recycle it by putting it back in the FILL ring (advertising the chunk to be used for next received packet) or, ⑥ if the packet needs to be forwarded, enqueue it in the TX ring.

Transmit path (user and kernel context). ⑥ The transmission path is as follows: user-space applications enqueue packets in the TX ring and then issue a syscall to kick start the transmission. ⑦ The actual transmission of the descriptors occurs in the kernel context, where similar to the receive path it is executed by a NAPI instance. ⑧ The NAPI instance will also reclaim the packets that were sent and will enqueue them in the COMPLETION ring. ⑨ The application can then poll the COMPLETION ring and reuse the UMEM chunks for new packets (advertise them to receive path or write data to them and transmit them).

2.2 OpenNIC

OpenNIC is an open-source network interface card code that compiles on AMD/Xilinx FPGAs [1]. OpenNIC serves as a foundation for building custom NIC offloads. Figure 3 shows the high-level overview of the NIC, composed of the Ethernet Media Access Controller (MAC) and Physical Coding Sublayer, which connects to QSFP cages and the AMD/Xilinx Multi-Queue DMA (QDMA) engine. Between these two components, developers can add their custom made offloads using native hardware description languages (i.e. Verilog or VHDL). Moreover, recent efforts have enabled the possibility to use use P4 [5] and eBPF [12] to build new offloads.

3 CHALLENGES SUPPORTING AF_XDP

Achieving high performance with AF_XDP requires support from the network interface card (NIC) driver. While it is possible to use AF_XDP without NIC driver support, this results in a fallback to copy mode, which significantly degrades performance [13]. Integrating NIC drivers with the AF_XDP kernel subsystem presents several challenges, which we discuss in this section. These challenges stem either from hardware-specific interfaces or from the limited visibility that the driver has into the behavior and requirements of the application running on top of it.

Resource	RSS	aRFS	Extra Resources
LUT	2809	2809	1×
LUTRAM	322	573	1.77×
Flip-flop	1964	5252	2.6×
BRAM	0	2	–

Table 2: *Resource utilization on the Xilinx/AMD Alveo U280 featuring the UltraScale+ XCU280 FPGA. The RSS module is implemented using the standard Toeplitz hash algorithm while the aRFS with a 64 entries CAM.*

(1) RSS and flow steering. XSKs are tied to a single UMEM, and a single queue (Figure 1). A UMEM can be shared between multiple queues (create a shared memory across queues in user-space program), but at the time of writing, the XDP program can only redirect traffic to XSK bound to the same hardware Rx queue (this was possible but later removed due to security concerns [24]). This effective limits software receive flow steering (RFS).

RFS is vital for the correct operation of AF_XDP based applications and also have grave implications on how their performance scale. If applications flow land on a hardware queue not bound to the XSK, XDP can not steer it to XSK and the packets are lost. The common solutions that we are aware of are:

Current options:

- *Single-queue mode:* The most straightforward but least scalable option is to use a single NIC queue and bind a XSK to it. This entirely avoids the complexity of steering, but it becomes a severe bottleneck and nullifies the scalability benefits of RSS. In this mode the steering is delegated to the user-space.
- *Socket multiplexing on one queue:* Multiple XSKs can share a queue, allowing different flows or roles to be handled by distinct sockets. However, this requires the XDP program to inspect and classify each packet in software to determine its destination socket. This adds overhead in the fast path, undermines performance, and bypasses the NIC's hardware steering capabilities.
- *Advanced hardware steering with aRFS:* Ideally, one would use a NIC with support for accelerated Receive Flow Steering (aRFS) or similar hardware-based flow steering. This allows dynamic mapping of flows to specific queues, which can then be served by zero-copy AF_XDP sockets with minimal overhead. Unfortunately, implementing aRFS support on programmable NICs such as OpenNIC requires resources which can no longer be used to implement application logic. In Table 2 we quantify this trade-off by comparing resources required to implement RSS and aRFS. The former is implemented in Hardware Description Language (HDL) using the standard Toeplitz hash algorithm [23]. The latter uses the Binary CAM LogiCORE IP from Xilinx/AMD [11] to implement a 64 entries Content Addressable Memory that matches flow IDs to specific DMA queues. We observe that in this

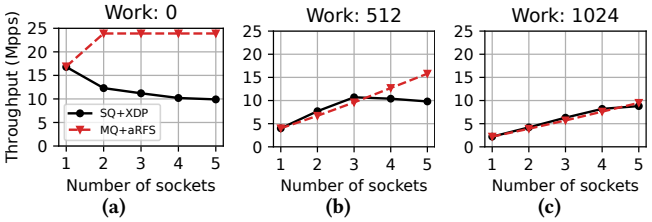
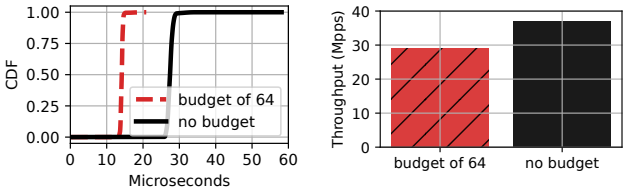


Figure 4: *Comparing socket multiplexing inside XDP or using aRFS. (**SQ:** single queue,**MQ:** multiple queue)*



(a) *Not bounding the Tx budget can reduce the applications responsiveness, because NAPI may take 10 µs longer.* **(b)** *Configuring transmit budget affects throughput of Tx heavy application*

Figure 5: *Linux kernel does not expose a Tx budget configuration while its value shows different trade-offs*

case aRFS requires 2.6× more flip-flops and 1.77× more LUTRAM. These resource requirements grow as we increase the number of aRFS rules that can be supported.

How in-kernel functionalities can help. Our observation shows that implementing aRFS might not be feasible or desirable on a programmable NIC. Most FPGA-based programmable NICs only provide a barebone programming framework, and do not implement features like aRFS out of the box. This forces the system developer to dedicate the limited resources of the FPGA to implement logic that is not related to the offloaded application. An alternative is to use eBPF and implement RFS in kernel [3]. In this setting, multiple XSKs are attached to a single queue (Figure 1), and a user supplies an eBPF program to determine to what XSK each packet is routed, enabling software multiplexing.

In Figure 4 we evaluate the performance of software multiplexing in XDP compared to aRFS. The test program receives packets through XSKs, finds a Fibonacci number as a proxy for packet processing complexity (the work parameter in the Figure 4), and finally sends the packet back. The number of XSKs changes from 1 to 5. Each XSK is owned by a different thread polling it constantly. With XDP, we observe contention when accessing FILL or COMPLETION rings creating a bottleneck for I/O heavy workload. With aRFS each socket is bound to a different hardware queue and avoids the contention, and throughput scales linearly. When no cycles are required to process packets, aRFS can achieve line-rate (25 Gbps for send and receive traffic in total) with two XSKs.

Of course, the current RFS implementation is limited: XSKs must be attached to a single queue, and thus only a single queue's traffic can be multiplexed across XSKs. This affects scalability, e.g., limiting the ability to use multiple queues (and thus cores) to process incoming flows that are distributed across queues using RSS. One way to improve this situation is to generalize Linux's RFS implementation and allow a single eBPF program to multiplex packets received from multiple queues across XSKs.

(2) TX descriptor posting and NAPI time-sharing. `AF_XDP` posts TX descriptors from within the NAPI poll loop, interleaving RX and TX processing. This is in contrast to the Linux's socket implementation where only RX processing is performed in the NAPI poll loop, and TX logic is implemented in the syscall context. While RX processing is bounded by the NAPI budget (default 64 packets), there is no universal TX budget, leading to either prolonged NAPI instances or TX ring bloat. The common approaches are:

Current options:

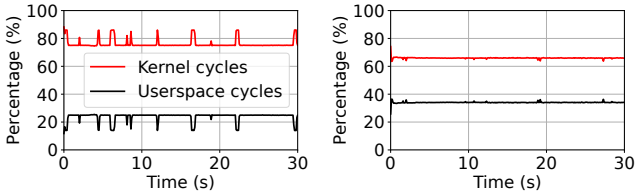
- *Fixed TX budget (e.g., 64 descriptors):* Mimic the RX budget to cap TX work per poll. This prevents a single NAPI invocation from monopolizing CPU time but may underutilize available TX ring space and reduce transmission burst efficiency.
- *Fill TX ring aggressively:* Post as many descriptors as the ring allows on each poll. This maximizes throughput and burst performance but risks excessively long NAPI polls that starve RX processing and can increase RX latency.

How in-kernel functionalities can help. In Figure 5, we show the trade-off between the two approaches. We ran a `AF_XDP` throughput generator, `txpush` [2], and measured the/ NAPI execution time and throughput. NAPI execution time was measured by taking timestamps at the start of the function and just before returning. Figure 5a shows that without any fixed budget applications that produces a lot of TX traffic can inflate the execution time of its NAPI instance by up to 90%, while experiencing an increase in throughput of 27%, pictured in Figure 5b. Prolonged NAPI instances lead to higher latencies in processing incoming packets, as the RX budget is fixed to 64 packets per NAPI execution.

The takeaway is that the TX budget should be configurable at runtime because the optimal value depends on workload characteristics. Moreover, the introduction of programmable NICs allows for complex offloads on the transmission path, that increase the processing time on the NIC and thus buffering. By configuring this parameter it would be possible to have a system level knob to control buffering on the transmission path of the offload and thus optimize for performance.

Our approach to achieving this builds on the observation that the kernel already monitors the execution time of NAPI instances to enforce an upper time bound and prevent system starvation caused by excessively long softirq execution [29]. If the kernel observes a low number of NAPI instances executed within a time interval, this may indicate that individual instances are consuming excessive time, potentially due to heavy XSK TX activity. Here, if one of the affected NAPI instances is zero-copy enabled, it may be beneficial to reduce its transmit budget to alleviate system pressure. One way to support this is to have an eBPF program that dynamically adjust the TX budget at runtime based on system policies.

(3) xsk_wakeup and hardware interrupt capabilities. The goal of `ndo_xsk_wakeup()` is to schedule a NAPI instance. Each driver has to implement its own version of this callback. The way the first drivers that supported `AF_XDP` did it was to write to a memory mapped register of the NIC that caused an interrupt. This way the NAPI instance was scheduled from the interrupt service



(a) *softirq* **(b)** *kthread*

Figure 6: *The lack of on-demand interrupts leads to the NAPI instance executed in softirq being scheduled on the same core of the application, starving it from clock cycles. With threaded NAPI enabled and pinned to the ISR's core the application can use more CPU.*

routine (as usual). A lot of NICs do not have this hardware feature[2] and resort to calling the function that schedules the NAPI from the process context of the application that issued the wakeup. Due to how the subsystem is designed, the polling function will be scheduled on the same core of the application. In order to achieve maximum performance the application and the NAPI instance should avoid timeshare on the same core as much as possible.

Current options:

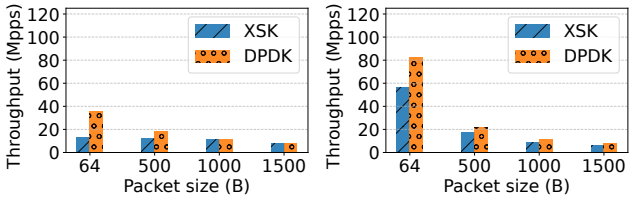
- *Threaded NAPI pinned to application core:* Use threaded NAPI, which is a mode of operation in which the NAPI poll function lives in a dedicated kernel thread and pin it to the IRQ line's CPU. This ensures wake-ups always occur on the same core, eliminating migrations. No driver changes are needed, but it introduces overhead of a kernel thread per queue and lacks comprehensive public performance evaluations.
- *Affinity-aware rescheduling within the driver:* At each NAPI reschedule, check the IRQ line's affinity mask; if the NAPI instance is on the wrong core, forcibly reschedule it to the application's core. This retains interrupt-based wake-ups without threaded NAPI, but adds the cost of affinity checks on every poll and incurs a scheduling delay when moving NAPI instances.

How in-kernel functionalities can help. To better understand this trade-off, we employed the same packet generator as in the previous point, on top of OpenNIC, that does not support MMIO based on-demand interrupts. We enabled TX completion IRQs, to have a source of interrupts to keep the NAPI on the interrupt service routine (ISR) core. Figure 6a illustrates how user-space execution is periodically starved of CPU cycles, as the softirq executing the NAPI polling context is scheduled on the same core as the user-space application, causing momentary drops in the produced throughput. In cases where NAPI executes on the intended core, it is due to an interrupt triggering execution on that core. In contrast, Figure 6b shows the performance when using a threaded NAPI instance pinned to the ISR core, resulting in improved isolation between kernel and user-space.

Recent efforts have introduced eBPF programmability in the Linux scheduler [6, 20] make it possible for custom scheduling policies that may alleviate this problem.

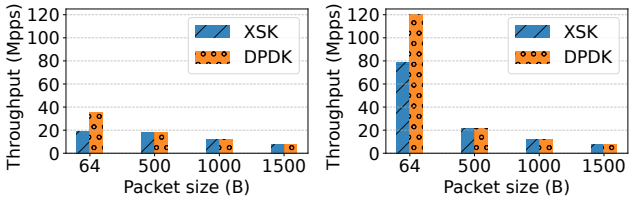
An eBPF-based scheduler could be utilized to prioritize the execution of the `ksoftirqd` thread responsible for the NAPI context, thereby reducing the latency associated with migrating NAPI instances across CPU cores. This prioritization can improve cache locality and overall responsiveness, particularly under high-throughput conditions. Furthermore, the scheduler could enable a dynamic execution model for `AF_XDP` applications, switching between single-core and dual-core configurations depending on system load. In the single-core mode, the `AF_XDP` application and its corresponding NAPI instance would be co-located on the same core to minimize context-switching and cache invalidation overhead. Conversely, in dual-core mode, they could be distributed across cores to leverage parallelism under high load. Implementing such a model would require a mechanism to expose execution mode preferences or runtime decisions to user-space applications, potentially through extended socket options or eBPF maps.

[2]This is because, as mentioned in point (1), hardware resource limits make it harder to implement complex functionality while meeting timing.



(a) *Single NIC's queue* **(b)** *Eight NIC's queues*

Figure 7: *L2 swap and forwarding benchmark comparing between DPDK and* `AF_XDP` *Sockets (XSK)*



(a) *Single NIC's queue* **(b)** *Eight NIC's queues*

Figure 8: *Packet drops benchmark comparing between DPDK and* `AF_XDP` *Sockets (XSK)*

4 XDP FOR OPENNIC

We have implemented a version of the OpenNIC driver that supports eBPF/XDP and `AF_XDP` zero-copy, in 1500 LoC [25].

Regarding the challenges outlined in Section 3: (1) the driver operates with either the RSS mechanism provided by the bare shell or the custom aRFS module shown in Table 2; (2) we opted to retain an unbounded TX budget by default, although this parameter remains configurable within the driver; and (3) while the latest version of the driver supports affinity-aware rescheduling, we prefer to run it with threaded NAPI and manually assigned CPU affinities, as this setup provides greater control and performance consistency.

In this section we report on our implementation's performance. Experiments were conducted on CloudLab [16] infrastructure hosted at the University of Massachusetts. The node we used is `fpga-alveo` which is equipped with dual-socket Intel Xeon Gold 6226R processors, each operating at 2.90 GHz. Hyper-threading and IRQ balancing were disabled to ensure consistent performance and reduce scheduling noise. The system was provisioned with 188 GB of RAM.

The node is equipped with a AMD/Xilinx Alveo U280 FPGA, on which the base shell of OpenNIC[1] was flashed.

We evaluate our implementation on two classes of stress tests: L2 Mac Address swap and forward in Figure 7 and packet drops rate in Figure 8. For AF_XDP it was used the xdpsock benchmark utility [2, 22] for both experiments; for DPDK testpmd was used for the L2 forwarding and dpdk-pktgen for the RX drop benchmark and as the line-rate packet generator for all tests. We test the performance using one and eight NIC's queues, which is maximum amount of queues currently supported in the base OpenNIC shell. At low packet sizes the performance mismatch between DPDK and AF_XDP is more noticeable and expected as seen in the literature [22, 26]. In single queue benchmarks (Figure 7a and 8a) DPDK outperforms AF_XDP by a factor of 2× and by 1.5× when using eight queues (Figure 7b and 8b). By increasing packet sizes, and thus reducing the how many packets per seconds are needed to reach line rate, AF_XDP can achieve very similar performance as DPDK, even by using only one queue.

Our measurements are comparable to prior AF_XDP measurements [22, 26] taken using commercial fixed-function NICs, showing that our implementation performs similarly.

5 RELATED WORK

The benefits of using AF_XDP in production environments have been discussed in the context of Open VSwitch [32], an industrial-grade software switch. In addition to this, the research community and beyond have proposed the use of AF_XDP to build high-performance DNS services [10], scalable 5G telecommunication runtime [30, 31], cloud gateways [33], and more efficient implementation of transport protocols [14, 19]. While all of those works are built on top of AF_XDP , many others focus on understanding its performance when used in combination with commodity NICs. Specifically, Karlsson *et al.* [22] studied the internals of AF_XDP implementation finding optimizations for better cache locality or simplifying communication if hardware supports in order delivery of descriptors; Parola *et al.* [27] analyzed the suitability of AF_XDP for different edge network function use-cases; Finally, Castillon du Perron *et al.* [15] looked at different configurations of AF_XDP to find the best parameters for reducing latency.

6 CONCLUSION

AF_XDP demonstrates that hardware support can enable zero-copy network libraries that are easier to use and deploy. But AF_XDP was designed to use widely-available NIC features. Our efforts have enabled and open-sourced [25] AF_XDP support for OpenNIC allow the community to consider other designs that split NIC and software functionality differently, and thus offer different tradeoffs. This work does not raise any ethical issues.

Acknowledgments. We thank our shepherd, Kahina Lazri, and other anonymous reviewers for their insightful comments. This work was partially supported by a research grant from NEC Laboratories Europe, and also partially funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Health and Digital Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

REFERENCES

[1] 2022. AMD OpenNIC Project. https://github.com/Xilinx/open-nic. (2022).
[2] 2022. Bpf-Examples/AF_XDP-example/README.Org at Main · Xdp-Project/Bpf-Examples. https://github.com/xdp-project/bpf-examples/blob/main/AF_XDP-example/README.org. (2022).
[3] 2022. Receive Flow Steering (RFS). https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/6/html/performance_tuning_guide/network-rfs. (2022).
[4] 2023. NAPI - The Linux Kernel documentation. https://docs.kernel.org/networking/napi.html. (2023).
[5] 2024. ESnet SmartNIC hardware design repository. https://github.com/esnet/esnet-smartnic-hw. (2024).
[6] 2024. Sched-Ext/Scx. https://github.com/sched-ext/scx. (2024).
[7] 2025. Accelerated Receive Flow Steering (aRFS). https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/6/html/performance_tuning_guide/network-acc-rfs. (2025).
[8] 2025. DPDK: Data Plane Development Kit. https://www.dpdk.org/about/. (2025).
[9] 2025. Fabric testbed. https://portal.fabric-testbed.net/. (2025).
[10] 2025. Improving DNSdist performance with AF_XDP. https://blog.powerdns.com/improving-dnsdist-performance-with-af_xdp. (2025).
[11] AMD. 2024. Content Addressable Memory (CAM). https://www.amd.com/en/products/adaptive-socs-and-fpgas/intellectual-property/ef-di-cam.html. (2024).
[12] AMD/Xilinx. 2023. The Nanotube Compiler and Framework. https://github.com/Xilinx/nanotube. (2023).
[13] Björn Töpel. 2018. AF_XDP: Introducing Zero-Copy Support. https://lwn.net/Articles/756549/. (2018).
[14] Zhongjie Chen, Qingkai Meng, ChonLam Lao, Yifan Liu, Fengyuan Ren, Minlan Yu, and Yang Zhou. 2025. eTran: Extensible Kernel Transport with eBPF. In *Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX.
[15] Killian Castillon du Perron, Dino Lopez Pacheco, and Fabrice Huet. 2024. Understanding Delays in AF_XDP-based Applications. (2024). arXiv:cs.NI/2402.10513 https://arxiv.org/abs/2402.10513
[16] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Annual Technical Conference (ATC)*. USENIX.
[17] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX.
[18] Dimitra Giantsidi, Julian Pritzi, Felix Gust, Antonios Katsarakis, Atsushi Koshiba, and Pramod Bhatotia. 2025. TNIC: A Trusted NIC Architecture: A hardware-network substrate for building high-performance trustworthy distributed systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM.
[19] Tianyi Huang and Shizhen Zhao. 2023. Accelerating QUIC with AF_XDP. In *International Conference Algorithms and Architectures for Parallel Processing (ICA3PP)*. Springer-Verlag.
[20] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *Symposium on Operating Systems Principles (SOSP)*. ACM.
[21] Intel. 2025. https://www.intel.com/content/www/us/en/developer/articles/technical/cloud-native-data-plane-leverage-af-xdp-kubernetes.html. (2025).
[22] Magnus Karlsson and Björn Töpel. 2018. The path to DPDK speeds for AF XDP. In *Linux Plumbers Conference*.
[23] Hugo Krawczyk. 1995. New Hash Functions for Message Authentication. In *Advances in Cryptology (EUROCRYPT)*. Springer.
[24] Greg Kroah-Hartman. 2024. CVE-2024-39293: Revert "xsk: Support redirect to any socket bound to the same umem". https://lore.kernel.org/linux-cve-announce/2024062548-CVE-2024-39293-d42a@gregkh/. (2024).
[25] Marco Molè. 2025. OpenNIC Driver with AF_XDP Support. https://github.com/system-fab/onic-afxdp. (2025).
[26] Jalal Mostafa, Suren Chilingaryan, and Andreas Kopmann. 2023. Are Kernel Drivers Ready For Accelerated Packet Processing Using AF_XDP?. In *Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE.

[27] Federico Parola, Roberto Procopio, and Fulvio Risso. 2021. Assessing the performance of XDP and AF_XDP based NFs in edge data center scenarios. In *International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*. ACM.

[28] Luigi Rizzo. 2012. netmap: a novel framework for fast packet I/O. In *Annual Technical Conference (ATC)*. USENIX.

[29] Rami Rosen. 2013. *Linux Kernel Networking: Implementation and Theory*. Apress.

[30] Bhavishya Sharma, Shwetha Vittal, and A Antony Franklin. 2023. FlexCore: Leveraging XDP-SCTP for Scalable and Resilient Network Slice Service in Future 5G Core. In *APNet*. ACM.

[31] Nishanth Shyamkumar, Piotr Raczynski, Dave Cremins, Michal Kubiak, and Ashok Sunder Rajan. 2022. In–Kernel Fast Path Performance For Containers Running Telecom Workload". https://netdevconf.info/0x16/sessions/talk/inkernel-fast-path-performance-for-containers-running-telecom-workload.html. (2022).

[32] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. 2021. Revisiting the Open vSwitch Dataplane Ten Years Later. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[33] Qianyu Zhang, Gongming Zhao, Hongli Xu, Zhuolong Yu, Liguang Xie, Yangming Zhao, Chunming Qiao, Ying Xiong, and Liusheng Huang. 2022. Zeta: A scalable and robust East-West communication framework inLarge-Scale clouds. In *Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX.