POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Driver Support for Programmable End-Host Networking

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Author: **Marco Molè**

Student ID: 220283
Advisor: Prof. Gianni Antichi
Co-advisors: Farbod Shahinfar
Academic Year: 2023-24

Abstract

High-performance packet processing is essential for modern networking, and eXpress Data Path (XDP) has become a key technology for accelerating packet handling in Linux. This thesis contributes to OpenNIC, an open-source FPGA-based Network Interface Card (NIC) with powerful offload and programmability capabilities, by enhancing network stack programmability through the integration of native support for XDP, a in-kernel eBPF hook for processing Ethernet frames in the driver, at earliest possible time. This work has also changed the memory model to Page Pool, a memory allocator tailored for the network subsystem, which has significantly improved performance, achieving up to a 200% increase in single-flow TCP throughput while also future-proofing the driver codebase for forthcoming enhancements in the kernel memory subsystem. Full stack programmability is achieved by the AF_XDP support, a userspace packet processing framework natively supported by the kernel that is built on top of XDP and specialized driver support.

Performance evaluation demonstrates that `XDP_DROP` achieves 50–80 million packets per second (Mpps) depending on packet access patterns, while `XDP_TX` and `XDP_REDIRECT` reach 30 Mpps and 37 Mpps, respectively. AF_XDP benchmarks show a peak drop rate of 50 Mpps and an L2 forwarding rate of 40 Mpps. By enabling both hardware offload within the NIC and software offload with XDP and AF_XDP, OpenNIC provides a flexible research platform for exploring hybrid packet processing strategies. This work establishes a foundation for future innovations in open-source NIC development and high-performance networking.

Keywords: End-Host Networking, XDP, SmartNIC, Programmable Networks

Abstract in lingua italiana

L'elaborazione dei pacchetti ad alte prestazioni è essenziale per il networking moderno ed eXpress Data Path (XDP) è diventata una tecnologia chiave per accelerare la gestione dei pacchetti in Linux. Questa tesi contribuisce a OpenNIC, una scheda di interfaccia di rete (NIC) open-source basata su FPGA con potenti capacità di offload e programmabilità, migliorando la programmabilità dello stack di rete attraverso l'integrazione del supporto nativo per XDP, un hook eBPF all'interno del kernel per l'elaborazione dei frame Ethernet nel driver, il prima possibile. Questo lavoro ha anche cambiato la gestione della memoria, introducendo al driver Page Pool, un allocatore di memoria pensato per il sottosistema di rete, che ha migliorato significativamente le prestazioni, ottenendo un aumento fino al 200% del throughput TCP a flusso singolo e rendendo il codice del driver a prova di futuro per i prossimi miglioramenti del sottosistema di memoria del kernel. La programmabilità dell'intero stack è ottenuta grazie al supporto AF_XDP, un framework per l'elaborazione dei pacchetti nello spazio utente supportato dal kernel e costruito sulla base di XDP e che richiede supporto specializzato nel driver.

La valutazione delle prestazioni dimostra che `XDP_DROP` raggiunge 50-80 milioni di pacchetti al secondo (Mpps) a seconda dei pattern di accesso ai pacchetti, mentre `XDP_TX` e `XDP_REDIRECT` raggiungono rispettivamente 30 Mpps e 37 Mpps. I benchmark AF_XDP mostrano un tasso di drop dei pacchetti di 50 Mpps e un tasso di inoltro L2 di 40 Mpps. Consentendo sia l'offload hardware all'interno della NIC che l'offload software con XDP e AF_XDP, OpenNIC fornisce una piattaforma di ricerca flessibile per esplorare strategie ibride di elaborazione dei pacchetti. Questo lavoro pone le basi per future innovazioni nello sviluppo di NIC open-source e di reti ad alte prestazioni.
Parole chiave: End-Host Networking, XDP, SmartNIC, Reti Programmabili

Contents

Introduction

The continuous performance gains predicted by Moore's Law and Dennard Scaling are reaching their physical limits. Transistor miniaturization is slowing due to fundamental constraints in power efficiency and heat dissipation, making it increasingly difficult to achieve higher clock speeds and processing power through traditional scaling methods. At the same time network speeds have continued increasing.

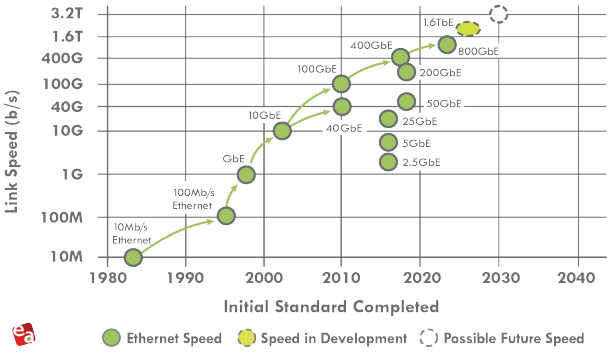


Figure 1: Evolution of Ethernet speeds

At 100 Gbps, with a 64-byte packet size, a 4 GHz CPU core has only 24 clock cycles to process each packet—an insufficient budget for any practical use case. As network speeds continue to increase, reaching 800 GbE [4] and even 1.6 TbE, this constraint will become even more severe, making it impractical for general-purpose CPUs to handle packet processing efficiently.

To address these challenges, networking architectures must move away from traditional, general-purpose hardware and software in favor of specialized processing offloaded to hardware, coupled with tailored software stacks optimized for high-speed data movement and minimal overhead. Technologies such as SmartNICs , DPUs, FPGAs, and in-network computing are emerging as essential solutions for accelerating packet processing, reducing latency, and offloading workloads from the host CPU [39] [61] [32] [66] [33] .

Achieving efficient data processing at these speeds requires a holistic approach, encompassing new hardware accelerators, optimized software stacks, and high-performance host

interconnects. The integration of RDMA, kernel bypass frameworks (e.g., DPDK and XDP), and dedicated hardware offloads will be crucial in ensuring scalable and efficient networking for next-generation cloud, AI, and HPC environments.

AMD OpenNIC [26] addresses this gap by providing an open-source, programmable SmartNIC that allows researchers and students to experiment with hardware offloads in a flexible, accessible environment. While OpenNIC has a maximum bandwidth of 100 Gbps, making it lower than cutting-edge commercial SmartNICs, it remains an excellent platform for understanding and developing next-generation networking solutions.

Designing hardware offloads requires developing accompanying software interfaces. XDP (eXpress Data Path) [46] is a crucial technology in this evolution. Operating within the Linux driver layer, XDP enables execution of fast packet processing by bypassing the traditional networking stack, significantly reducing CPU overhead.

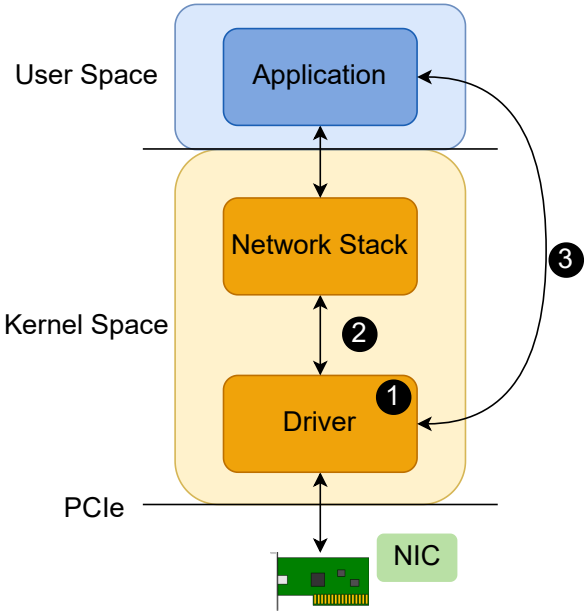


Figure 2: **1** XDP is a eBPF hook for processing Ethernet frame executing in the driver context, **2** Page Pool allows the driver and the network stack to recycle packets buffers and improve performance **3** AF_XDP is a in-kernel zero-copy fast path between user space and the driver for raw packet processing.

Contribution

The contributions of this work are: ① extending on extending the AMD/Xilinx OpenNIC Linux driver by adding full XDP support; ② integrating support for a new memory allocator, called Page Pool, which optimizes buffer management for high-speed packet

processing by reducing memory allocation overhead and improving cache efficiency; ③ added support for `AF_XDP`, a zero copy path that enables for efficient and custom user-space packet processing by eliminating unnecessary memory copies that is implemented by leveraging portions of the kernel's XDP subsystem. Figure 2 shows how each of these subsystems are either fully encompassed in the driver or are bridging the driver with either the kernel stack or userspace.

The XDP support and Page Pool contribution has been merged into the master branch of the official OpenNIC driver [15], making it accessible to the broader research and developer community. With this integration, users of OpenNIC can now experiment with XDP, leveraging its low-latency processing capabilities for network offloading, high-speed packet filtering, and custom networking applications. This work provides a foundation for further research into hybrid software-hardware acceleration. At the time of writing no other open source FPGA based SmartNICs have native XDP support. By enabling XDP on OpenNIC, this work bridges the gap between programmable hardware offloads and high-performance software networking, fostering further innovation in low-latency, high-speed networking solutions. The integration of `AF_XDP` is public but not merged because the bare OpenNIC shell lacks hardware flow steering, making this feature too advanced for its current hardware capabilities.

1 | Background

1.1. Linux Drivers

The role of drivers in operating systems is to bridge together the interfaces offered by the operating system and the one provided by the hardware they are written for. Linux distinguishes three classes of device drivers: *Character devices*, *Block devices* and *Network devices*. I'll briefly touch on the first two to provide a more complete taxonomy, then I will go in detail on how network drivers are structured in Linux and what are the kernel subsystems that are used by them.

1.1.1. Char & Block Drivers

Character devices represent hardware that processes data as a stream of characters, operating on a sequential basis. These devices permit direct, unbuffered access to the underlying hardware and do not provide a random-access capability. Typical examples include serial ports, keyboards, mice, and various sensors. The character device model creates a representation in the /dev directory, where each device is accessible through file operations. The fundamental operations—open(), read(), write(), and close()—map directly to the corresponding system calls. This implementation exemplifies the Unix philosophy, as users and applications can interact with hardware using standard file I/O paradigms.

Block device drivers manage hardware that transfers data in fixed-size blocks rather than individual bytes. These devices, primarily storage media such as hard disks and SSDs, provide random access capabilities and generally employ caching mechanisms to optimize performance. The block device subsystem inserts an additional layer of abstraction—the block I/O layer—between the filesystem and the physical device. This architecture enables efficient caching through the page cache, request queueing and merging to optimize physical I/O operations and support for diverse filesystems while maintaining hardware independence.

Despite this additional complexity, block devices still maintain adherence to the file abstraction principle. They appear as files in /dev and support standard I/O operations,

though optimized for block-level access:

1.1.2. Network Drivers

Network interfaces provide an interesting exception to the UNIX design principle of *everything is a file*, as they do not appear in the */dev* directory nor they implement the *struct file_ operations* that registers the callbacks for typical file operations like *read(), write()* and *seek()*. Where char devices and block devices work only at the request of the kernel, network devices have to invert this relationship and prompt the kernel to notify it about incoming packets.

1.1.3. DMA in Linux

Direct Memory Access (DMA) is a technology that allows non-CPU hardware components to write directly to the RAM. This is the foundational mechanism that all NIC use to send/receive packets and metadata about the status of the RX/TX queues. The setup of a DMA region involves coordinating between the virtual kernel address space, the physical memory addresses, and the *bus address*. The bus address is the view of the memory space from the device point of view. The bus address could match the physical one, but in modern systems there is a dedicated MMU to IO devices, the IOMMU. The IOMMU improves security and compatibility, but causes a significant decrease in performance [35] [37] [28].

Handling DMA shared memory in SMP architectures forces the developer to distinguish between buffers that have to be *coherent* at all time between the CPU and the Device, thus by disallowing caching those addresses, and buffers that can be cached by the CPU. An example for the latter case are buffers that support one-way transmissions and thus can be cached by the CPU once received, but it requires the developer to use explicit synchronization instructions to make sure that the CPU is not seeing stale data. Linux calls them *coherent* and *streaming* buffers [8].

Coherent buffers are expensive to use due to the fact that they inhibit all sorts of caching. They guarantee only that the memory operations will be immediately visible in the RAM, not that instruction will be reordered by the CPU, so the use of memory barriers has not to be precluded. It is the form of DMA used by the data structures used for communicating where packets should be DMAed by the NIC and how many packets has the OS processed: *descriptor rings* and *completion rings*.

Streaming buffers are where the packets are DMAed by the NIC. In the receive path, the

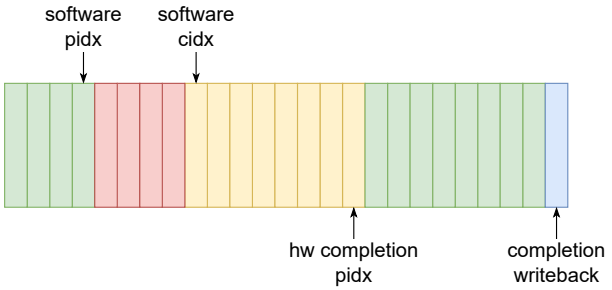


Figure 1.1: Red are empty descriptors slots, yellow is for valid descriptors pointing to data to process and green is for valid descriptors pointing to empty pages. Bottom indexes are written by the QDMA engine, while top indexes by the driver

driver has to explicitly sync the buffer for the CPU and in the transmission path it has to sync it for the device.

1.1.4. Descriptor and Completion rings

Transmission and reception of packets between the NIC and the Host happens via multiple transmission and receive and queues, allowing parallel processing of packets at the software level. Each queue has it's own interrupt line and its own replica of the notification handling mechanism that runs in *ksoftirq* threads: NAPI, described in section 1.1.5.

Receive queues

Each receive queue is implemented using a pair of single-producer, single-consumer DMA-mapped ring buffers: one for descriptors and one for completions. Additionally, a control structure, referred to as the context, manages these buffers.

The descriptor ring stores references to allocated pages, with the software acting as the producer and the hardware as the consumer. Once the driver allocates the pages and populates the descriptors, it updates the producer index (`PIDX`), which marks the end of the valid region. This index is then written to the corresponding context via memory-mapped I/O (MMIO).

The completion ring contains metadata about incoming packets, with the hardware serving as the producer. In the case of the QDMA, the DMA implementation used by OpenNIC, completion entries include the packet length, a color bit for distinguishing valid regions when the buffer wraps around, an error field, and an incremental ID. The last slot of the buffer, known as the writeback completion status, communicates the hardware's completion producer index (`PIDX`) to the driver [10].

The receive path follows these steps:

1. When the interface is initialized, the driver allocates n pages, maps them as streaming DMA, and populates the descriptor ring with their bus addresses.
2. The driver updates the context's software producer index (`sw pidx`) to n and the software consumer index (`sw cidx`) to 0 via MMIO.
3. Upon receiving a packet, the NIC retrieves a valid descriptor (the green slots in Figure 1.1) and performs a DMA transaction to the corresponding address.
4. The NIC then updates the hardware producer index (`hw pidx`) and, if enabled, generates an interrupt.
5. The interrupt handler schedules the NAPI instance associated with the queue.
6. The NAPI polling function reads the hardware producer index (`hw pidx`) from the writeback slot and processes the pending packets, which reside in the yellow region delimited by `sw cidx` and `hw pidx`.

When the amount of unused valid descriptors (the green region) gets below a critical threshold the driver has to allocate new buffers, post their DMA address in the descriptor ring and communicate the new `sw pidx`.

Transmission queues

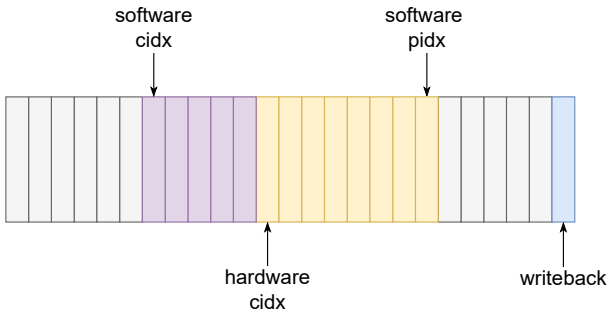


Figure 1.2: Purple slots are to-be reclaimed buffers, yellow slots are packet not yet sent by the NIC. The blue slot is the writeback status.

The transmission queues are implemented using a single descriptor ring, with the last slot of the array reserved as a writeback status slot. In this slot, the driver reads the hardware consumer index (`hw cidx`), which indicates the last packet sent by the NIC.

The driver maintains two pointers: the *software producer index* (`sw pidx`), representing the last packet written to the NIC's transmit queue, and the *software consumer index* (`sw`

`cidx`), which, together with the *hardware consumer index* (`hw cidx`), allows the driver to determine which packets have been transmitted. This information enables the kernel to reclaim and free the corresponding buffers.

The transmission process follows these steps:

1. The driver writes packet descriptors to the transmit queue, advancing the software producer index (`sw pidx`).
2. The driver notifies the NIC of new packets by updating the corresponding context via MMIO.
3. The NIC fetches the packet descriptors and transmits the packets.
4. Once transmission is complete, the NIC updates the hardware consumer index (`hw cidx`) in the writeback status slot.
5. The driver reads `hw cidx` and advances the software consumer index (`sw cidx`), reclaiming all buffers within the purple region of Figure 1.2.
6. The yellow region in Figure 1.2 represents packets that have been written to the transmit queue but have not yet been sent by the NIC.

1.1.5. NAPI: IRQ then poll

Once incoming packets are stored in memory pages, the next challenge is how to notify the host of their presence efficiently. The goal is to design a flexible system that prevents the host from being overwhelmed with interrupts (IRQs) during high traffic loads while maintaining low latency for time-sensitive flows.

The subsystem responsible for handling packet reception notifications in Linux is called NAPI[1]. It combines the two classical notification mechanisms: interrupts and polling. The key idea behind NAPI is to initiate polling for packets only after an interrupt has been received. Polling continues until the host detects that the queue is empty. The polling function operates in the **softirq** context, a mechanism for handling high-priority but deferrable tasks outside the immediate scope of a hardware interrupt. SoftIRQs process network packets, timers, and other asynchronous events without blocking the system for extended periods. When a softirq is raised, it is typically executed within the same CPU context as the interrupt handler, i.e. before the context is restored to the interrupted task. However, if execution takes too long, it is deferred to a per-CPU kernel thread called `ksoftirqd`. This creates a good balance of performance and fairness in the system

[1]NAPI originally stood for "New API".

because kernel threads have to compete with other threads for CPU time and user space does not get starved[31].

The driver has to 1) create a `napi_struct` for each of its receive queues, 2) Register one interrupt line for each queue and in the IRQ handler call the `napi_schedule` function, that will insert the reference of the `napi__struct` into a work queue, 3) Implement and register onto the kernel the function that will perform the polling.

In the OpenNIC driver the polling function is `int onic_rx_poll(napi_struct *napi, int budget)`, it will process packets, up to `budget` per call. The return value is number of packets actually processed. If a single execution exhaust the budget the kernel will guarantee that that napi instance will be re-scheduled right after. This allows the driver to safely disarm the interrupt line for that queue, and avoid the extra cost of handling unnecessary IRQ hanlders executions. If the function does not exhaust the budget the driver will re-arm the interrupt line, and the NAPI instance will be rescheduled only after a new IRQ.

In the polling function the driver also performs the reclaiming of TX descriptor described in 1.1.4. [16] [60] [47]

1.2. eBPF

eBPF (Extended Berkeley Packet Filter) is a powerful and flexible technology in the Linux kernel that allows developers to run custom, sandboxed programs at various kernel execution points without modifying the kernel source code or requiring module recompilation. Initially designed for packet filtering, eBPF has evolved into a general-purpose tool for networking, observability, and security tasks. The eBPF programming model involves writing programs in a restricted subset of C, Go or Rust, which are then compiled into bytecode and loaded into the kernel via user-space utilities like bpftool or libraries such as libbpf[13] or aya[2]. These programs are verified for safety by the kernel's verifier, ensuring they terminate and access only permitted resources. Once loaded, eBPF programs attach to hooks like tracepoints, kprobes, network sockets, or eXpress Data Path (XDP) layers. They are widely used for performance monitoring, tracing, security policy enforcement, and accelerating packet processing, offering both high performance and significant flexibility while maintaining kernel stability.

The appeal of eBPF is that it allows kernel customization, without having the effort of maintaining either a fork of the kernel or an out-of-tree module. These two methods have their valid uses but are complicated to maintain synced with upstream sources and third

party forks could introduce critical bugs hard to debug without the help of the Linux open source community. To mitigate this latter point the kernel allows execution of eBPF object files only after they have passed an in-kernel static verifier that check the safety of the program. A more detailed analysis of the verifier can be found in [42], in summary it checks:

- no out-of-bound memory access, no use after free and more in general the integrity of the pointers used,
- type safety of the available kernel memory accessible to the eBPF program,
- potential resources leaks by the program, like missing freeing of memory or releases of locks,
- termination of the program, as otherwise the eBPF program could starve the kernel,
- absence of deadlocks,
- absence of data race with the kernel, all the interactions the program can have with kernel resources are mediated by specific *helpers* that implem

Having these restrictions on the execution model has forced a strict codification of the interfaces available to the eBPF programs that are used to communicate across eBPF programs, to kernel space and to user space.

eBPF Maps: Safe Shared State

eBPF maps provide a structured and verifier-checked way to share data between eBPF programs and between eBPF and user-space applications. They serve as the primary means of maintaining state across multiple invocations of an eBPF program. Several types of maps exist, including:

- Array and hash maps, which store key-value pairs with strict type constraints.
- Per-CPU maps, which optimize performance by reducing contention on shared data.
- Ring buffers and perf buffers, which allow efficient event passing between eBPF and user space.

The verifier ensures safe access to maps by enforcing bounds checking and verifying proper key/value usage, preventing corruption of kernel memory. Figure 1.3 shows the lifecycle of a eBPF program with a user space control plane, both compiled with Clang, and how the can communicate via the maps. The eBPF program can only access these data structures via pre-defined functions called Helper functions.

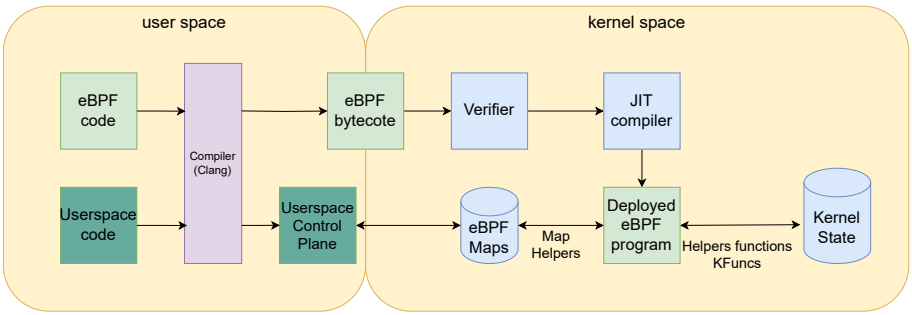


Figure 1.3: eBPF programs can access stateful maps, occasionally shared with user space, and Kernel state via predefined Helper functions.

Helper Functions & KFuncs

Functions accessible by eBPF programs have to be carefully written as they're not checked by the verifier, and must take care in not exposing sensible information or crash the system. The difference between Helpers and KFuncs is stability guarantee across kernel versions, Helpers have the same guarantees of UAPI (the userspace API), ensuring that they will likely never change semantic or disappear. KFuncs do not have this guarantees and can be dynamically registered by kernel modules, allowing for a extending the functionality of eBPF runtime with custom routines.

F

1.2.1. XDP - eXpress Data Path

The eXpress Data Path is the eBPF hook positioned at the reception of the Ethernet frame in the NIC driver. The program is run before the `sk_buff` allocation, the structure that encapsulates packet data along with metadata, such as protocol headers, timestamps, and routing information, providing a flexible and extensible framework for network processing. Programs are attached to a network interface and replicated for each queue of that interface. Same instances of the same program will run concurrently in each queue NAPI instance, so when programming XDP programs with shared state the developer has to either shard state with `*_per_cpu` maps or with eBPF spin locks. Having different XDP programs per RX-queue was in the original design, but was reject by the upstream because it required the NIC to support per-flow steering rule (aRFS) [6].

The eBPF program has access to packet and can read it, modify it and produce a verdict about its future [20]:

- `XDP_PASS`: pass it to the kernel network stack for normal processing,

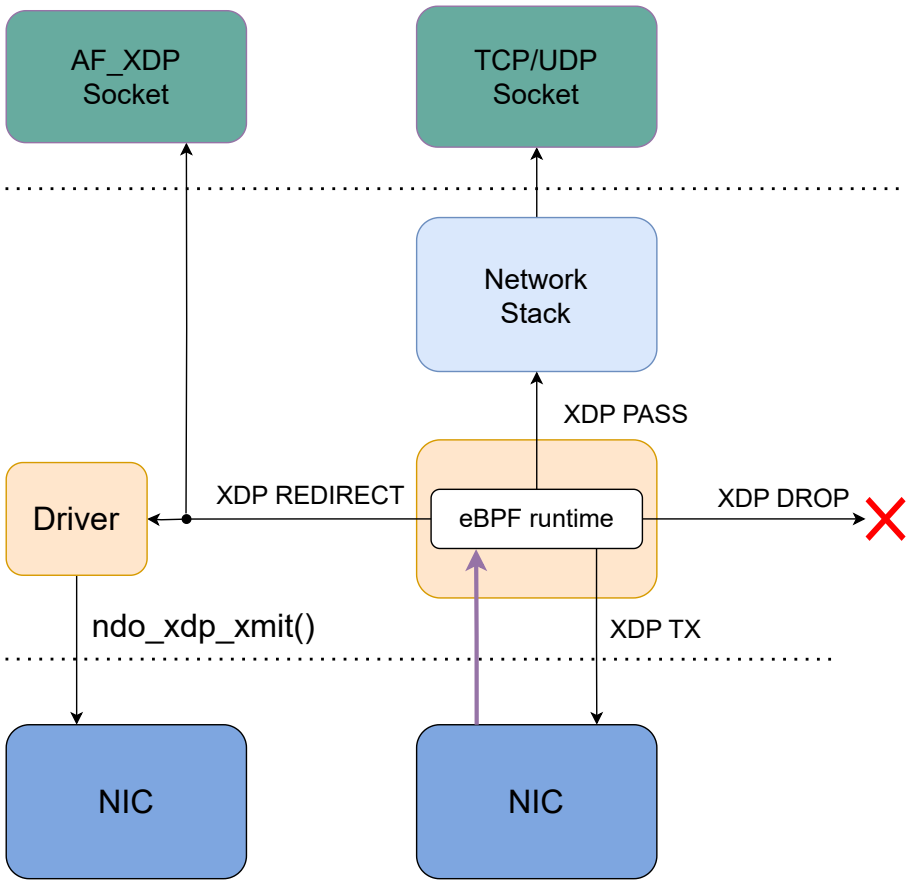


Figure 1.4: The purple arrow represents the incoming flow of packets, where each packet triggers the execution of the eBPF/XDP program attached to the corresponding network interface.

- `XDP_DROP`: drop the packet before `sk_buff` allocation,
- `XDP_ABORTED`: like above, but it will produce a visible kernel tracepoint, `xdp_exception`,
- `XDP_TX`: transmit the packet out of the same NIC it arrived from,
- `XDP_REDIRECT`: redirect the packet to other NICs, eBPF programs running on specific CPUs or special userspace socket called AF_XDP.

Figure 1.4 illustrates the action of each described verdict, demonstrating that the in-driver programmability introduced by XDP can be highly expressive in terms of supported operations.

```
1 SEC("xdp")
2 int xdp_pass_func(struct xdp_md *ctx) {
3   bpf_printk("Hello␣world!");
4   return XDP_DROP;
5 }
```

Listing 1.1: *Hello world* program that prints to a in kernel log and then drop the packet

Listing 1.1 shows a very basic XDP program that prints a string into a kernel buffer accessible at `/sys/kernel/tracing/trace_pipe`, and then drops the packet. This will stop all flows arriving at the interface on which this program is attached to. The `SEC("xdp")` macro marks the `xdp_pass_func` function has the entry point of the XDP program. The `struct xdp_md *ctx` contains the metadata about the packet to process like

- `ctx->data`: Pointer to the start of the packet data,
- `ctx->data_end`: Pointer to the end of the packet data,
- `ctx->data_meta`: Pointer to optional metadata,
- `ctx->ingress_ifindex`: Interface index on which the packet arrived,
- `ctx->rx_queue_index`: The queue index of the NIC where the packet was received

The `data` and `data_end` pointers are especially important when writing XDP programs because they delimit the valid region of memory accessible by the eBPF runtime, and each memory read or write of the packet must be checked against these two pointers. In Listing 1.2 there are two functions, the first for parsing the Ethernet header and the second to parse the IP header of the packet. As we can see in line 6 and 19 before accessing the packet's data it's a pretty common pattern to 1) cast a pointer to the struct that represents the header, 2) compute the size of header, that in some cases (IP,TCP, etc) could have a variable size due to options fields thus requiring a double check of the

boundaries, first for the memory region that contains the fixed part of the header, and then for the variable part.

```
static __always_inline int parse_ethhdr(void *data, void *data_end,
                                        __u16 *nh_off, struct ethhdr **ethhdr) {
    struct ethhdr *eth = (struct ethhdr *)data;
    int hdr_size = sizeof(*eth);

    if ((void *)eth + hdr_size > data_end)   // without this check the verifier
        return -1;                           // will reject this function

    *nh_off += hdr_size;
    *ethhdr = eth;

    return eth->h_proto; /* network-byte-order */
}

static __always_inline int parse_iphdr(void *data, void *data_end,
                                       __u16 *nh_off, struct iphdr **iphdr) {
    struct iphdr *ip = (struct iphdr *)(data + *nh_off);

    if ((void *)ip + sizeof(*ip) > data_end)
        return -1;

    // ihl is the number of 32-bit words in the header, so we multiply by 4 to
        get the number of bytes.
    // ip header has a variable length, so we need to check
    // that the header is fully contained in the packet
    int hdr_size = ip->ihl * 4;
    if (hdr_size < sizeof(*ip))
        return -1;
    if ((void *)ip + hdr_size > data_end)
        return -1;

    *nh_off += hdr_size;
    *iphdr = ip;

    return ip->protocol;
}
```

Listing 1.2: Example of two functions parsing network headers in the eBPF programming style of checking the validity of every pointer before accessing it

The XDP hook is peculiar because it requires deep driver support, as its verdict have to be implemented working directly with the descriptor rings which are not exposed with a common interface to the network stack of the kernel.

In the kernel is present a fallback XDP hook, called XDP generic, which is executed by the kernel after `sk_buff` allocation and thus requires no driver support. This hook is for compatibility mode and does not have the speed gains XDP provides. There has been also work regarding offloading the eBPF runtime inside smartNICs,

In section 2.1 I'll show how each verdict is implemented in the context of the ONIC driver.

Moreover, XDP's design has been extended with the introduction of AF_XDP [1] (commonly referred to as xsk or XDP sockets), which allows user-space applications to efficiently interact with XDP programs. AF_XDP enables zero-copy data paths by leveraging memory-mapped rings shared between user space and the kernel, minimizing CPU usage and reducing latency. However, implementing xsk also requires additional support from the driver, as the zero-copy mode demands compatibility with both XDP and user-space buffer management.

Additionally, the emergence of multibuffer support has added further complexity. While XDP was originally designed for single-buffer packet processing, multibuffer enables handling larger packets that exceed the size of a single buffer, such as jumbo frames, or NICs that split header and data in two different descriptors. To support multibuffer functionality, the driver must be capable of managing and chaining multiple buffers while still maintaining compatibility with XDP's high-performance constraints.

XDP has found a lot of industrial and academic success, as a lot of hyperscalers, cloud providers and CDN companies are using it in critical infrastructure for L4 layer load balancing [19] [22], DDOS mitigation [12] and deep observability of their networks, both virtual and physical [3].

1.3. AMD OpenNIC project

The OpenNIC project consist of a "open source"[2] FPGA-based NIC shell, the Linux driver and a DPDK driver.

[2]QDMA and CMAC IP are not open source

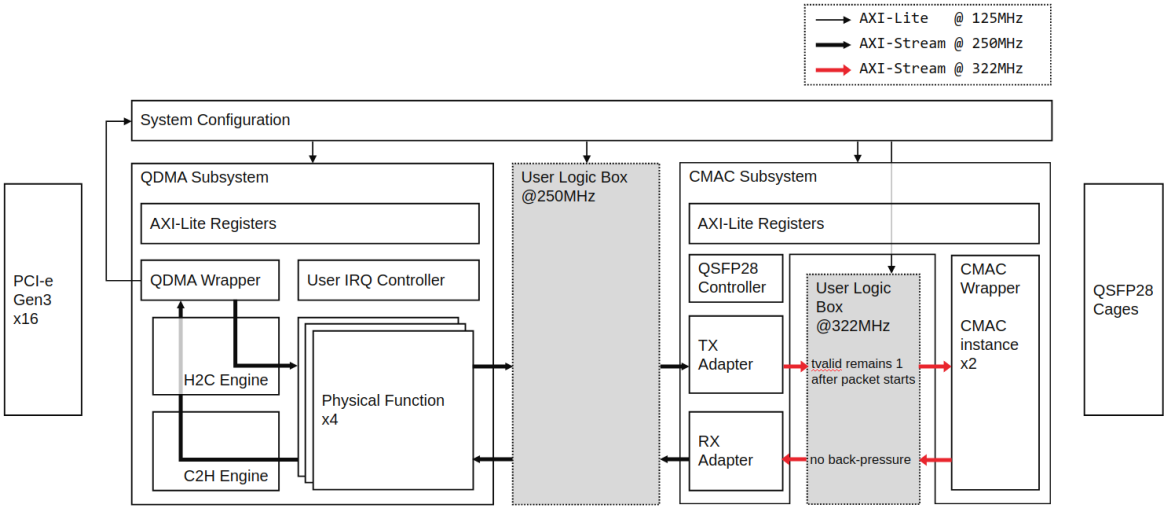


Figure 1.5: High level architecture of the OpenNIC shell. [26]

The shell skeleton consists of several key components:

- QDMA Subsystem – Contains Xilinx QDMA IP and bridging logic, interfacing with a 250MHz user logic box via a custom AXI4-stream protocol.
- CMAC Subsystem – Includes Xilinx CMAC IP with wrapper logic, supporting 1 or 2 CMAC ports. It operates at 322MHz and interfaces with user logic using a 322MHz AXI4-stream protocol.
- Packet Adapter – Converts between the 250MHz and 322MHz AXI4-stream protocols, acting as a packet-mode FIFO. It also restores back-pressure capability on the RX path.
- System Configuration – Handles reset mechanisms and register allocation using a 125MHz AXI4-lite interface.

There are two user logic boxes running at 250MHz and 322MHz, each with an AXI-lite interface for register access and AXI4-stream interfaces for TX/RX. User RTL plugins manage these interfaces

It enables research on cutting-edge networking technologies by allowing users to implement custom packet processing logic within the two User Logic (UL) blocks. These programmable hardware blocks provide a high degree of flexibility, enabling the development of network functions, data processing pipelines, and software/hardware offloading mechanisms. Researchers can leverage these capabilities to explore novel architectures, optimize packet processing workflows, and experiment with innovative approaches to net-

work acceleration.

In addition to its general-purpose programmability, OpenNIC also includes a variant that supports Remote Direct Memory Access (RDMA), a widely adopted technology in modern datacenter networks. RDMA enables high-throughput, low-latency communication by allowing direct memory access between devices, bypassing the CPU and operating system. This feature makes OpenNIC a valuable bridge between academic research and industry-grade networking technologies, providing a practical platform for testing and prototyping advanced network mechanisms [68].

The importance of working with FPGA-based hardware offloads cannot be overstated, especially given the growing disparity between network bandwidth advancements and CPU performance improvements. While network speeds continue to increase rapidly—driven by the demand for high-performance cloud computing, distributed storage, and AI workloads—traditional CPU architectures struggle to keep pace. Offloading computationally intensive networking tasks to programmable hardware, such as FPGAs, helps mitigate this bottleneck by accelerating packet processing and reducing CPU overhead. This shift toward hardware acceleration is becoming increasingly critical in achieving scalability and efficiency in modern high-speed networks.

2 | Architecture

In this chapter I will show how XDP is integrated at the driver level first for the original fixed buffer memory model and then for the novel page pool memory allocator. I'll argue on why the fixed buffer approach is a bad fit for the XDP execution model both for performance and memory handle.

2.1. XDP Support

The interaction between the driver and the kernel happens via a struct of function pointers, a pattern seen frequently in the Linux codebase, called `net_device_ops`. It serves as an interface for network device drivers to implement specific methods for device initialization, packet transmission, configuration, and management. For setting up a XDP program we have to register the following callback: `int (*ndo_bpf)(struct net_device *dev, struct netdev_bpf *bpf)`. The `netdev_bpf` struct has a `command` field that specifies one of the following actions: mounting or unmounting of a XDP program, setting up a queue for XSK mode or perform a hardware offload of the XDP program. Since OpenNIC does not support hardware offloading I'll not cover this latter commands. The command for setting up XSK mode will be shown in Section 2.3.

The `XDP_SETUP_PROG` command signals to the driver can access `bpf->prog` which contains the reference to the actual XDP program. The driver has to copy this reference to all queues. By copying the reference the each different queue we reduce the number of global state read by each CPU, since each NAPI instance runs on different cores.

Another key component of the per-queue state is the `xdp_rxq_info` struct that holds metadata that are used by the kernel to keep track from which queue xdp buffers have origin and which memory model the queue is using. This is done to ensure that the XDP subsystem knows how memory is managed when it has to handle `XDP_REDIRECT`. The reason for having multiple memory models in XDP is that different network workloads and hardware configurations require different approaches to memory management. Traditional network stacks rely on standard kernel mechanisms for allocating and freeing memory, but

XDP introduces optimizations that allow for significantly higher packet processing speeds. To achieve this, XDP supports distinct memory models that dictate how packet buffers are allocated, accessed, and recycled.

- The `MEM_TYPE_PAGE_ORDER0` model is designed for cases where memory allocation in single-page chunks (order-0 pages) is preferable. In Linux memory management, order-0 pages are the smallest unit of memory allocation, corresponding to a single page of RAM.
- The `MEM_TYPE_PAGE_SHARED` model is the conventional reference-counted approach, where memory pages are shared and tracked using a reference counter. This model aligns with standard kernel memory management practices, ensuring compatibility with existing networking code. However, reference counting introduces some overhead, which can be a bottleneck in high-performance packet processing scenarios.
- The `MEM_TYPE_PAGE_POOL` model is specifically optimized for XDP, offering a more efficient way to handle packet buffers. The page pool mechanism allows for fast recycling of memory pages, reducing the need for expensive memory allocations and deallocations. This model improves cache efficiency and minimizes the overhead associated with traditional memory management, making it ideal for high-speed packet processing.
- The `MEM_TYPE_XSK_BUFF_POOL` model is designed for `AF_XDP`, a mechanism that enables zero-copy packet transfer between the kernel and userspace applications. This memory model allows packets to be directly placed in a buffer pool that userspace applications can access, avoiding unnecessary copies and improving performance for high-speed packet processing in applications like DPDK (Data Plane Development Kit).

The OpenNIC driver started with a `MEM_TYPE_PAGE_ORDER0` model but the pivoted to the new page pool (Section 2.2) for all queues by default and the ability convert individual queues to zero copy xdp (Section 2.3). First I will show how to implement XDP on the `MEM_TYPE_PAGE_ORDER0` model, then how to migrate the whole driver to `MEM_TYPE_PAGE_POOL`.

Before running the eBPF virtual machine the driver has to construct `xdp_buff`, a metadata structure that holds pointers delimiting the parts of the buffer in which the packet is present. It must live on the stack for minimizing overheads.

- `data`: Pointer to the start of the packet data,
- `data_end`: Pointer to the end of the packet data,

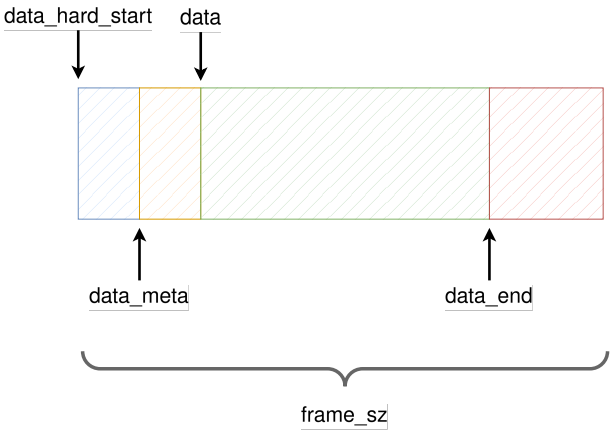


Figure 2.1: `xdp_buff` describes the data layout of the packet to the XDP runtime.

- `data_meta`: Pointer to metadata associated with the packet,
- `data_hard_start`: Pointer to the hard start of the packet buffer,
- `rxq`: Pointer to the XDP receive queue information,
- `frame_sz`: Size of the buffer holding the packet, generally set to PAGE_SIZE [1]

The blue region in Figure 2.1 is called *headroom* and it's standardized to be 256 bytes, even though drivers could choose to make smaller or bigger. Its purpose is to enable XDP programs to add headers to packet, like VLAN or IP-in-IP tunneling, without having to pay the cost of *memmoveing* the entire payload. The yellow region is used for metadata [2]. XDP Metadata serves two different use cases: accelerating XDP programs with hardware computed metadata or accelerating the other eBPF programs up the processing stack (like TC) with information produced at the XDP level. Metadata coming from the hardware can be accessed with Kfuncs, thus more can be added dynamically to suit specific needs. The green region is the actual packet, from the start of the Ethernet header to the end of the application layer payload. Whenever the XDP program wants to access a region of the packet it has to check it is inside the boundaries of `data` and `data_end`. The red region is called *tailroom* and can only be accessed after invoking an helper function that moves the `data_end` pointer, enlarging the memory available to the XDP program.

If the driver constructs this structure with erroneous data the eBPF verifier will allow the program to do malign actions, like overwriting others packet in the same buffer.

Before explaining how to handle the verdicts I will explain how the memory model

[1] Some NICs, notably Intel's, put more packets into one page as a memory saving technique. In those cases frame_sz should be the logical amount devoted to each packet in a page

[2] Recently various members of the XDP community have been advocating for a name change to XDP Hints

`MEM_TYPE_PAGE_ORDER0` worked in the context of the OpenNIC driver, as it is necessary to understand how memory is handled before adding the XDP logic. I describe the memory handling as a *fixed descriptor* approach, in which the descriptor rings are filled with descriptors at driver initialization with pointers to allocated and mapped pages. In the receive path the drivers allocates a `sk_buff` and a *new* page in which it copies the packet received in the DMA mapped buffer.

```
1   for ( min(64, ring_distance(hw_pidx, sw_cidx)){
2       buf = /* buffer of the current packet */
3       struct xdp_buff xdp_buff;
4       dma_sync_single_for_cpu(&priv->pdev->dev,
5                               buf->dma_addr + buf->offset,
6                               len, DMA_FROM_DEVICE);
7       xdp_init_buff(&xdp_buff, PAGE_SIZE, &q->xdp_rxq);
8       xdp_prepare_buff(&xdp_buff, page_address(buf->pg), buf->offset, len,
            true);
9       res = onic_run_xdp(rx_queue, &xdp_buff);
10      if (res & ONIC_XDP_REDIR) xdp_xmit = true;
11      if (res & ONIC_XDP_PASS){
12          skb = napi_alloc_skb(napi, pkt_len);
13          skb_put_data(skb, page_address(buf->pg) + buf->offset, len);
14          ...
15          ret = napi_gro_receive(napi, skb); // Pass up to the network stack
16          ...
17      }
18       if (res & ( ONIC_XDP_TX | ONIC_XDP_REDIRECT))
             onic_refill_page(rx_queue);
19  }
20  ... //just before returning from the napi instance
21  if (xdp_xmit) xdo_do_flush();
22  ...
```

Listing 2.1: Core of the NAPI polling function the fixed buffers. `napi_alloc_skb` allocates the `sk_buff` and a blank page fragment of length *pkt_len*. `skb_put_data` performs a *memcopy* from the DMA buffer to the newly allocated data portion of the skb.

On line 4-6 it synchronize the buffer for the CPU because it utilizes streaming mappings, and on line 7 and 8 it fills the `xdp_buff` as described before.

XDP_PASS

The `XDP_PASS` verdict means the packet has to be passed up to network stack, and handling it is no different than handling the reception of packets in a non XDP scenario. Lines 11 to 15 showcase in order: `sk_buff` and new page allocation with the `napi_alloc_skb` function, the population of data done with `skb_put_data` and the ownership change from the driver to the network stack with `napi_gro_receive`.

XDP_DROP

The `XDP_DROP` verdict signals that the packet has to be dropped immediately, blocking any further processing. Handling drops in the fixed buffers memory model is very easy because it just skips the portion of code that allocates the `sk_buff`, going directly to the next packet.

XDP_TX

The `XDP_TX` verdict means that the XDP program has decided that the packet has to be transmitted out the interface it came from. Implementing `XDP_TX` requires re-implementing the same logic that is used for traditional transmission of packets but with the `xdp_buff` as the metadata structure, instead of `sk_buff`.

The implementation consist of two functions: `onic_xdp_xmit_back` converts the `xdp_buff` into a `xdp_frame`, another metadata structure used to represent packet inside the XDP kernel subsystem, and `onic_xmit_xdp_ring` which posts the descriptor into the transmission queue. The conversion makes so that the second function can be used by the driver when it is the target of a xdp redirect. The `xdp_frame` is a metadata structure that lives in the headroom of the page (the blue region of Figure 2.1) that makes the packet self described, it also overcomes the fact that `xdp_buff` has to live on the stack and thus it's lifetime is tied to the NAPI execution that processed that packet.

When posting descriptors to the transmission queue there could be a race condition with the kernel that is trying to transmit on the same queue. This could lead to corrupted descriptor or queue state. To avoid this a spinlock has to be taken with `__netif_tx_lock(queue, cpu)` before calling `onic_xmit_xdp_ring`.

At this point should be clear how implementing `XDP_TX` completely breaks the fixed descriptors approach, because it's not possible to put a descriptor on both in the transmission queue and receive queue. The receive queue could write to the descriptor overwriting or corrupting the data to a page that may be in midst of a transmission. The solution is

that, in the case of TX (and redirect) a new page has to be allocated and the descriptor has to replaced, shown briefly at line 18 of Listing 2.1.

XDP_REDIRECT

The `XDP_REDIRECT` action is a very general action that could target very different object inside the kernel. An XDP program can redirect frames to other NICs, another XDP program running on a specific CPU or to a special "raw" socket in userspace. To manage the complexity derived by the diverse nature of the targets the XDP kernel subsystem subdivides the redirect action in three steps. The first action is to choose the redirect target in the XDP program and communicate it to the kernel. This is done by calling the helper function `long (* const bpf_redirect_map)(void *map, __u64 key, __u64 flags)` passing the map and the key representing the target.

```
1  struct {
2      __uint(type, BPF_MAP_TYPE_DEVMAP);
3      __uint(max_entries, 4);
4      __type(key, __u32);
5      __type(value, __u32);
6  } tx_port SEC(".maps");
7
8  SEC("xdp")
9  int xdp_redirect_func(struct xdp_md *ctx) {
10     __u32 index = 1; // Redirect to port with index 1
11     return bpf_redirect_map(&tx_port, index, 0);
12 }
```

Listing 2.2: Minimal example that shows how a XDP program uses the redirect helper function

The only action performed by `bpf_redirect_map` is to write the target information into `bpf_redirect_info` which lives in the `task_struct` of each process[3].

The second action is performed by the driver when it receives the `XDP_REDIRECT` verdict and it consist in calling the `xdp_do_redirect` function. This converts the `xdp_buff` into a `xdp_frame` thus passing the ownership of the packet's buffer to the kernel which then reads `bpf_redirect_info` and with that information enqueues the frame into map type-specific bulk queue structure. At the time of writing this queue structure has a fixed size of 16 packets, if redirect occurs on a full queue it gets flushed by the kernel.

[3]It used to be a per-cpu variable but it has changed after the PREEMPT-RT was changed as it

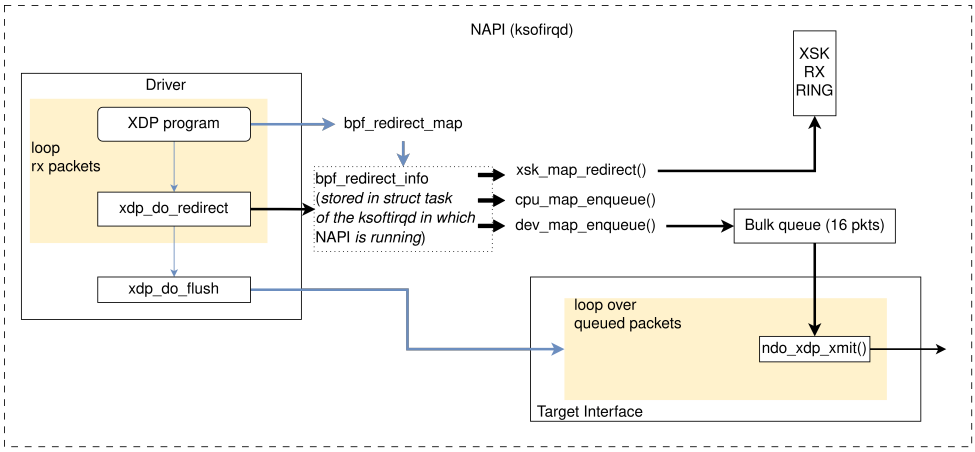


Figure 2.2: The blue arrows represent the control flow, black arrows represent packet flow.

The third and final step occurs in the driver, at the end of the NAPI poll function, and it's the invocation of the `xdp_do_flush` function at line 21 of Listing 2.1 , which flushes the queues and in the case of redirection to another interface calls sets the `XDP_XMIT_FLUSH`. It's critical that `xdp_do_flush` is called before `napi_complete_done` for two reasons: the redirect bulk queues live along side the `bpf_redirect_info` in the task struct, thus it essential to guarantee it's called on the same core as the `xdp_do_redirect` as the napi instance could be migrated to another ksoftirq when in idle mode; the second reason is that reading the map entries requires RCU protection, which is guaranteed in NAPI because the code is execute between `local_bh_disable()/local_bh_enable()`. [4]

The other side of redirect: ndo_xdp_xmit() For an interface to be a valid target for redirection, it must implement the callback function `int (*ndo_xdp_xmit)(struct net_device *dev, int n, struct xdp_frame **xdp, u32 flags)` within the `net_device_ops` structure. In OpenNIC, the callback implementation invokes the `onic_xmit_xdp_ring` function n times, following the same structure described for `XDP_TX`. At the time of writing, the only valid flag is `XDP_XMIT_FLUSH`, which indicates whether the driver must update the hardware with the new value of the software producer index (*pidx*). This mechanism is necessary because, under high load conditions, bulk queues are flushed when they reach their capacity limit. To mitigate the overhead associated with writing to the MMIO register, updates occur only when the flush flag is set.

If the target interface is unable to enqueue a packet in its transmission (TX) queue—either due to errors in the DMA mapping process or because the queue is full—it must abort the transmission of all remaining packets. The function returns the number of packets

required extra locks in the NAPI context [17]

[4] `local_bh_*()` are read-side critical-section markers [21].

successfully transmitted. If an error occurs and the returned value is smaller than n, the kernel is responsible for freeing the remaining packets. This behavior introduces a potential issue in the design of XDP programs: there is no guarantee that a redirected packet has been successfully transmitted.

2.2. Page Pool

The Linux kernel has introduced a specialized page allocator tailored for network drivers supporting XDP capabilities called Page Pool. As the name implies it handles not only allocation but the recycling of the pages after the packet has been consumed by either the XDP runtime or network stack. The driver has to allocate one pool for each receive queue, to avoid having contention and synchronization issues between queues when allocating or recycling. Internally the page pool is composed by a 128-slots array that acts as a lockless LIFO queue (*fast/lockless cache*) and by a `ptr_ring` set to the size of the receive ring that acts as a FIFO queue (*slow/locking cache).* `ptr_ring` is data structure holding pointers provided by the kernel designed to facilitate efficient communication between a single CPU producer and another single CPU consumer in high-performance, low-latency contexts while minimizing cache contention and synchronization overhead and it is used by the page pool when the recycling of the page happens outside of the NAPI context, thus potentially in a race condition with the NAPI itself trying to allocate pages.

Changing the memory model in the context of OpenNIC requires getting rid of the *fixed buffer* approach described before. When processing a packet its content are not copied to a fresh allocated page but instead the metadata structures are built with references to that page, requiring the driver to refill the descriptor ring with new pages.

I'll now describe how to create and configure the page pool, then how the allocation and recycling processes work and how they are related to implementing XDP verdicts. At the initialization of the queue the page pool is created with the following parameters

```
struct page_pool_params pp_params = {
    .flags    = PP_FLAG_DMA_MAP | PP_FLAG_DMA_SYNC_DEV,
    .order    = 0,
    .pool_size = ring_size,
    .nid      = dev_to_node(&priv->pdev->dev),
    .dev      = &priv->pdev->dev,
    .dma_dir  = xdp_prog ? DMA_BIDIRECTIONAL : DMA_FROM_DEVICE,
    .offset   = XDP_PACKET_HEADROOM,
    .max_len  = priv->netdev->mtu + ETH_HLEN,
```

```
10      .napi     = &rx_queue->napi
11  };
12
13  q->page_pool = page_pool_create(&pp_params);
14  ...
15  err = xdp_rxq_info_reg_mem_model(&q->xdp_rxq, MEM_TYPE_PAGE_POOL,
        q->page_pool);
```

Listing 2.3: Page Pool configuration as of Linux v6.8

At line 2 with the first flag it's specified that the page pool will be in charge of un/mapping pages (in the context of the device specified at line 6). Since the pages will have a streaming mapping (Section 1.1.3), they require explicit synchronization, the second flag guarantees that the pages will be automatically synced to the device by the pool internals. The fields at line 7,8,9 are used to determine exactly what portion of the buffer has to be automatically synced. Synchronization to the CPU has to be done by the driver. At line 3 it's specified the page's order, line 4 specifies the size of the `ptr_ring` and line 5 the NUMA node of the device. Specifying the NUMA node guarantees that the pages will be allocated from the same node, ensuring faster memory accesses. At line 10 it links the page pool to the NAPI instance of the queue. This is useful because it allows the network stack to check if the core in which it is freeing the page is the same in which is scheduled the NAPI instance, ruling out any possibility of race conditions and thus allowing for safe recycling in the *fast cache* even if outside the NAPI context. After creating the page pool the driver has to mark the queue with the `MEM_TYPE_PAGE_POOL` model and a reference to the pool itself. This will be used by the redirect subsystem to recycle pages after they have consumed by the target entity.

The allocation path follows the green arrows of Figure 2.3: (A.1) Initially the page pool examines the lockless cache; if it is found empty (A.2), the pool attempts to replenish it using pages recycled within the ring. If even the ring is empty the pool does a bulk allocation of 64 pages directly into the *fast cache* (A.3), and then the page is given to the driver.

Returning a page from a NAPI context involves the following process: (R.1) the page pool initially tries to recycle the page into the cache; (R.2) if this is not feasible, it attempts to return the page to the ring; (R.3) if this also fails, the page is then released from the pool back to the kernel. The Page Pool API is tailored for XDP where each page is used singly by a buffer. Thus, pages can be recycled only if the page is singularly referenced and the page has not been allocated from the critical pfmemalloc reserves. Pages that are

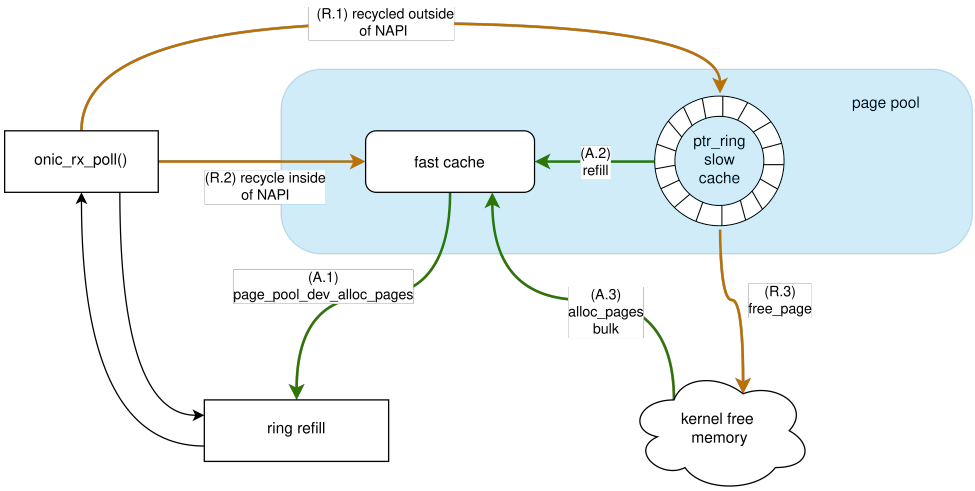


Figure 2.3: Page Pool internals

recycled in a non-NAPI context, i.e. pages that are recycled by the networking stack in the application context thread, (R.2) go directly in the ring.

Another advantage of the Page Pool is that it handles the DMA mapping for all the pages, using the parameters provided by the driver at initialization, and most importantly it keeps the pages DMA mapped as long they are tied the pool, avoiding the extra cost of un/mapping every page.

2.2.1. Handling XDP verdicts with page pool

Changing the memory model requires modifying the way XDP verdicts are handled, taking care of correctly recycling the pages.

XDP_PASS

`napi_build_skb` at line 2 of Listing 2.4 allocates a `sk_buff` that points to the start of buffer page. `skb_mark_for_recycle` flags the skb as containing a page that has to be recycled to a page pool. `skb_reserve` moves the data pointer of the skb to the actual start of the Ethernet frame, skipping the empty headroom. `skb_put` writes to the skb the length of the packet.

```
1   for ( min(64, ring_distance(hw_pidx, sw_cidx)){
2       .... // buffer preparation is the same
3       if (res & ONIC_XDP_PASS){
4           skb = napi_build_skb(xdp.data_hard_start, PAGE_SIZE);
5           skb_mark_for_recycle(skb);  // mark the skb for page_pool recycling
6           skb_reserve(skb, xdp.data - xdp.data_hard_start);  // reserve space
```

```
                in the skb for the data for the xdp headroom
7           skb_put(skb, xdp.data_end - xdp.data);  // set the data pointer
8           ...
9           ret = napi_gro_receive(napi, skb); // Pass up to the network stack
10          ...
11      }
12      ...
13  }
14  // .. regardless of the xdp verdict
15  onic_ring_refill(rx_queue);
```

Listing 2.4: Core of the NAPI polling function with the new memory model. `napi_build_skb` only allocate memory for the `sk_buff`, that is then constructed around the original page

Line 15 is executed regardless of the XDP verdicts after all the packets have been processed, and it refill the RX ring with new pages, allocated from the page pool.

XDP_TX

Adapting `XDP_TX` requires changing the logic on how the pages are reclaimed once transmitted. The reclaiming is done in a lot of places inside the kernel: when the kernel does normal transmission, when the interface is the target of a XDP redirect, during the tear down of the device (it happens on XDP attach and detach) and during the NAPI polling function. Only during the latter is possible to recycle into the lockless cache.

```
1  for (i = 0; i < work; ++i) {
2          struct onic_tx_buffer *buf = &q->buffer[ring->tail];
3          if (buf->type == ONIC_TX_SKB) {
4              dma_unmap_single(&priv->pdev->dev, buf->dma_addr, buf->len,
                   DMA_TO_DEVICE);
5              dev_kfree_skb_any(buf->skb);
6          }  else if (buf->type == ONIC_TX_XDPF) {
7              if (napi_direct) xdp_return_frame_rx_napi(buf->xdpf)
8              else xdp_return_frame(buf->xdpf);
9          } else if (buf->type == ONIC_TX_XDPF_XMIT) {
10             dma_unmap_single(&priv->pdev->dev, buf->dma_addr, buf->len,
                   DMA_TO_DEVICE);
11             xdp_return_frame(buf->xdpf);
12         }
13         onic_ring_increment_tail(ring);
```

```
14 }
```

Listing 2.5: Core loop of the reclaiming function for TX descriptors, `onic_tx_clean`

Listing 2.5 shows how the driver has to keep track of all the possible origins of the TX descriptors because each type requires different memory handling. Lines 6-8 handle the reclaiming of descriptors that originated from a `XDP_TX` verdict, marked with the `ONIC_TX_XDPF` type. The `xdp_return_frame` function is offered by the kernel and it either frees or recycles the page according to the `rxq` field present on the xdp frame. On line 7, if the caller of the reclaiming function could guarantee being inside the NAPI instance that generated the packet, the recycling will occur directly to the lockless cache. Later I'll discuss why the same cannot be done for redirect case which is handled on the next lines.

XDP_REDIRECT and ndo_xdp_xmit()

As with the previous verdict the main change regards the recycling logic after packets have been transmitted. In the `ndo_xdp_xmit()` function the driver has to map the frames in the DMA region of the device and mark them with the `ONIC_TX_XDPF_XMIT` type . This is why at line 10 of Listing 2.5 it has to manually unmap the page. This is not done for the TX case because one of the design choice of Page Pool is to keep the pages DMA mapped as long as they are linked to the pool. At line 11 the page is returned, if the source interface also uses Page Pool then the page will be recycled back into the slow cache of its pool. `xdp_return_frame` knows how to correctly free or recycle frames thanks to the `xdp_rxq_info` that tags each buffer/frame.

XDP_DROP

The Listing 2.6 shows the immediate recycling of the pages in the case of drop/aborted or errors in retransmissions.

```
1 static int onic_run_xdp(struct onic_rx_queue *q, struct xdp_buff *xdp_buff){
2     ...
3     act = bpf_prog_run_xdp(q->xdp_prog, xdp_buff);
4     switch(act){
5         case XDP_PASS: return ONIC_XDP_PASS;
6         case XDP_TX:
7             err = onic_xdp_xmit_back(q, xdp_buff);
8             if (err) goto out_error;
9                 return ONIC_XDP_TX;
```

```
10          case XDP_REDIRECT:
11              err = xdp_do_redirect(q->netdev, xdp_buff, xdp_prog);
12              if (err) goto out_error;
13  out_error:
14          case XDP_ABORTED:
15              trace_xdp_exception(q->netdev, xdp_prog, act);
16              fallthrough;
17          case XDP_DROP:
18              page_pool_recycle_direct(q->page_pool, page);
19              return ONIC_XDP_CONSUMED;
20      }
21  }
```

Listing 2.6: This is the function wrapping the call to the eBPF runtime. Failures or drops cause the driver to immediately recycle the page

I've chosen to show also the other switch branches to display how failures in other verdicts, full transmission ring or wrong redirect target, are handled in the `XDP\_DROP` case with a direct recycle into the lockless cache.

2.3. Zero Copy AF_XDP

`AF_XDP` is a socket type that allows userspace applications to work directly on Ethernet frame data. An XDP program can redirect ingress frames to a specific `AF_XDP` socket, and the application can read the frames directly from the socket. This can be done in a zero-copy manner, which can significantly reduce the number of memory copies and context switches between the kernel and userspace. This can be especially useful for high-performance networking applications, such as packet filtering, load balancing, and DDoS mitigation. The zero copy mode requires driver support as it involves working with a specialized memory pool that coordinates between the kernel and userspace. The `AF_XDP` socket type is available in the Linux kernel version 4.18 and later. `AF_XDP` sockets are also referred to as `XSK`. This should not be intended as a kernel bypass architecture like DPDK but rather as a *fast path* inside the kernel, useful for implementing custom and novel network protocols or specialized networking applications.

2.3.1. AF_XDP internals

`AF_XDP` operates through a set of specialized data structures that facilitate the transfer of packets between the user space and the kernel. The primary components include:

UMEM

UMEM (User Memory) is a contiguous memory region allocated in user space, specifically designated for storing network packets. This memory is divided into fixed-size chunks, each capable of holding a single packet. UMEM serves as the foundational buffer pool for both packet reception and transmission.

Descriptor Rings

`AF_XDP` utilizes multiple ring buffers to efficiently manage packet flow. These rings serve different roles in the processing pipeline:

- Fill Ring (FR): Managed by user space, this ring holds descriptors referencing free UMEM chunks that are available for receiving incoming packets.
- Receive Ring (RX): Managed by the kernel, this ring contains descriptors corresponding to packets that have been received and placed into UMEM chunks. These packets are ready for processing in user space.
- Transmit Ring (TX): Managed by user space, this ring holds descriptors pointing to UMEM chunks containing packets prepared for transmission.
- Completion Ring (CR): Managed by the kernel, this ring contains descriptors of successfully transmitted packets, signaling to user space that the corresponding UMEM chunks can be reused.

RX and TX ring are tied the socket and Fill and Completion rings to the UMEM. Each socket can be bound to only one UMEM but multiple sockets can share one UMEM. The Receive Ring is filled when an XDP program does a `bpf_redirect_map` with the file descriptor of the XSK. Looking at the internals of the XDP subsystem, Figure 2.2, the RX ring is filled by the `xsk_map_redirect` function executed in the flush step.

A XSK can be bound to only to a single NIC's queue, for this reason a "complete" `AF_XDP` system should support *advanced Receive Flow Steering* (aRFS), the ability to specify to the NIC per-flow steering rules. Ideally one would redirect the flows that should be processed by the XSKs on the zero-copy enabled queues, and steer away all the other flows.

An application can chose between two modes of operations. In the *run-to-completion (RTC)* model, the application continuously polls the Rx ring to check for incoming packets, processing them as soon as they arrive. This approach minimizes latency but can be CPU-intensive. In contrast, the *poll()* model allows the application to sleep when idle and only

wakes up when packets are available or a timeout occurs, making it more efficient in terms of CPU usage. For transmission, packets are placed in the Tx ring, and the kernel sends them either via a sendmsg() syscall or within the NAPI context. If the Tx ring is full, the application can either busy-wait (RTC model) or use poll() to wait for available space. In the RTC model each application consumes two core, one for the user space logic and one for the kernel processing, while the polling method works by stacking everything on a single core. This distinction does not affect how the driver implements`AF_XDP` operations.

Packet Processing Workflow

The interaction between UMEM and descriptor rings ensures efficient, zero-copy packet processing. The flow of packets is outlined as follows:

Packet Reception User space populates the Fill Ring with descriptors pointing to available UMEM chunks. The driver fills the descriptor ring with descriptors coming from the Fill Ring. The packet data is DMAed directly into the UMEM chunks referenced in the Fill Ring and subsequently updates the Receive Ring with the corresponding descriptors. User space retrieves packet descriptors from the Receive Ring and processes the incoming packets.

Packet Transmission User space writes data into UMEM chunks and places corresponding descriptors into the Transmit Ring. The kernel reads these descriptors, transmits the packets, and moves them to the Completion Ring upon successful transmission. User space monitors the Completion Ring to identify transmitted packets and reclaim the associated UMEM chunks for reuse.

2.3.2. Implementing AF_XDP support in OpenNIC

Setting up the pool The pool is created in the userspace application and then tied to the NIC's queue. The kernel invokes the `ndo_bpf` callback, with a new command: `XDP_SETUP_XSK_POOL`. This command has two fields: `xsk.pool` and `xsk.queue_id`, respectively a pointer to the XSK Buffer Pool that will serve as the memory allocator and the queue that the userspace is requesting to convert.

The driver has to check that the requested queue exists, and if it does it performs the DMA map of the pool in the context of the device. `xsk_pool_dma_map(pool, &priv->pdev->dev, DMA_ATTR_SKIP_CPU_SYNC);`

Then the driver tears down the desired queue and it brings it up as a zero copy queue.

The main differences from a normal page pool backed queues are: it uses the XSK Buffer Pool as it sole page allocator and thus is registered with the `MEM_TYPE_XSK_BUFF_POOL` memory model, and it ditches the buffer metadata structure used for page pool in favor of a plain array of `*xdp_buff`. Indeed, when allocating from the XSK pool, the function returns prefilled `xdp_buff`. The driver can use the `xsk_buff_xdp_get_dma` function to retrieve the DMA mapping of `xdp_buff` coming from the pool.

XSK support requires a new `net_device_ops` callback: `ndo_xsk_wakeup (struct net_device *dev, u32 qid, u32 flags)`. This function is utilized to activate the the task in which the NAPI instances of this queue run, that holds the responsibility for transmitting and/or receiving packets on a particular queue identifier associated with an `AF_XDP` socket. This callback is essential because the transmission of XSK packets happens inside the NAPI instance, and without the ability to manually wakeup the instance the transmission of packets would occur only in presence of inbound traffic. In theory the wakeup function should look something like this

```
1  int onic_xsk_wakeup(struct net_device *dev, u32 qid, u32 flags){
2  // Performs sanity checks
3  ...
4  // If the napi instance is currently running, change its state to MISSED.
5  // This will force the kernel to schedule it again immediatly.
6  if (!napi_if_scheduled_mark_missed(napi)){
7      // If the NAPI instance is not scheduled, trigger an IRQ using hardware
           mechanisms.
8      }
9  }
```

Listing 2.7: This is the function wrapping the call to the eBPF runtime. Failures or drops cause the driver to immediately recycle the page

Triggering the IRQ using hardware mechanism guarantees that the NAPI instance will run on the same core as before, especially important since in this kind of pseudo-bypass systems is not uncommon to fix interrupt lines and tasks to specific CPUs to have stable and predictable latencies and to avoid the potential performance losses induced by the `irqbalance` kicking in. Unfortunately, the OpenNIC shell does not support this. The necessary condition for generating queue interrupts is that a completion entry gets written to a completion ring of a receive queue. I speculate this could be implemented in the 250MHz User Logic box by injecting a dummy packet in the desired queue.

Instead, the real implementation does a direct invocation to `napi_schedule()` which puts

the NAPI instance in the poll list of the CPU on which the wakeup was issued, and not the one on which it was running before. This is not optimal for performance as it could lead to high cache misses.

Receive Path The processing of packets occurs as always in the NAPI polling loop, that has to be adjusted to handle `xdp_buff` backed by UMEM pages. At line 3 of Listing 2.8 the driver has to add the only missing field of the `xdp_buff`: the `data_end` pointer. The length of the packet is retrieved from the completion entry as usual. At line 4 the driver synchronizes the DMA buffer for the CPU, note how it's not the same function used in Listing 2.1 but instead is a specialized function that retrieves the DMA mapping from the internal state of the pool. The XSK driver's API are designed to mask any reference to the actual pages coming from the UMEM, reducing the risk for drivers to mishandle memory.

```
1  for ( min(64, ring_distance(hw_pidx, sw_cidx)){
2      struct xdp_buff *xdp_buff = q->xdps[desc_ring->tail];
3      xdp_buff->data_end = xdp_buff->data + len;
4      xsk_buff_dma_sync_for_cpu(xdp_buff, q->xsk_pool);
5      xdp_result = onic_run_xdp_zc(q, xdp_buff);
6      xdp_xmit |= xdp_result;
7      if (xdp_result == ONIC_XDP_CONSUMED) {
8          xsk_buff_free(xdp_buff);
9      }
10     else if (xdp_result == ONIC_XDP_PASS){
11         skb = onic_xsk_construct_skb(napi, xdp_buff);
12         err = napi_gro_receive(napi, skb);
13         ...
14         xsk_buff_free(xdp_buff);
15         }
16     }
17 ... // after all packet have been processed
18 if (xdp_xmit |= ONIC_XDP_REDIR) xdp_do_flush();
```

Listing 2.8: Core of the NAPI loop processing zero copy packets

At line 8, when the packet is dropped either intentionally or due to errors, `xsk_buff_free` gets called. This function does not actually *free* the page but recycles it in the *free list* of the XSK, which is analogous in scope to the *lockless cache* of the Page Pool. On line 5 and 11 other two functions that deviate from the default poll loop: `onic_run_xdp_zc` and `onic_xsk_construct_skb`. Handling XDP verdicts and constructing *skb*s has to change

again, as they are directly tied to how memory is handled.

XDP_PASS The driver cannot build the `sk_buff` around the page as it does in the Page Pool queues as the pages coming from the UMEM cannot be referenced outside the XSK lifecycle. `sk_buff` is constructed with `napi_alloc_skb` and `skb_put_data` as in Listing 2.1, then the `xdp_buff` is freed back to the UMEM.

XDP_TX `XDP_TX` is not considered a primary component in AF_XDP, as its implementation cannot be realized in a zero-copy manner due to the design limitations of the rings. During the conversion of an `xdp_buff` into an `xdp_frame`, a standard procedure across all versions of `XDP_TX` implementation, the kernel provides a built-in function `xdp_convert_buff_to_frame`. Specifically, for XSK `xdp_buff`s, this function copies them to a newly allocated, independent page, subsequently returning the `xdp_buff` to its original pool.

XDP_REDIRECT Redirect is handled gracefully by the XDP subsystem depending on the target: if the buffer is being redirected to an existing XSK socket then the kernel enqueues the packet in the corresponding RX ring. For any other target the aforementioned `xdp_convert_buff_to_frame` gets called, the contents copied to a fresh page and the `xdp_buff` is recycled back into the XSK pool.

XDP_DROP As always drop could be either intentional or due to errors in the TX/Redirect process, in any case the buffer is recycled at Line 8 of Listing 2.8.

Transmit Path The classic network stack transmission happens via the `ndo_start_xmit` callback, while `ndo_xdp_xmit` is used when transmitting `xdp_frame` coming from other interfaces. The transmit path of `AF_XDP` is a bit peculiar because it does not have a standalone point of entry for transmitting packets, but instead is performed inside the NAPI instance before processing incoming packets. To avoid any race condition and complex state management a TX XSK queue should only transmit XSK packets, thus the driver has to communicate to the kernel that the queue is stopped, so that it won't be used for skb transmission, and has also to avoid the queue in the `ndo_xdp_xmit` function.

The path is the following: `xsk_tx_peek_desc` is used to fetch descriptors to transmit. If the UMEM is shared between multiple sockets it will fetch the descriptors from all of them, in a round robin way. `xdp_desc` is a new data structure that serves as a proto-descriptor, containing basic information like the page address and the length of the packet. The driver then has to convert it into the specific format of descriptor used by the hardware. This

step is not shown in Listing 2.9 for brevity's sake. `xsk_tx_peek_desc` does not actually modifies the global state of the queue, it only updates the a local pointer to the ring. The `xsk_tx_release` function updates the global state by copying the cached value, that has to happen with a store release operation [14].

```
int onic_xsk_xmit(struct onic_private *priv, struct onic_tx_queue *q, int
    budget){
    ...
    struct xdp_desc xdp_desc;
    int trasmitted = 0;
    while (budget--){
        if (!xsk_tx_peek_desc(q->xsk_pool, &xdp_desc)) break;
        trasmitted++;
        dma_addr = xsk_buff_raw_get_dma(q->xsk_pool, xdp_desc.addr);
        xsk_buff_raw_dma_sync_for_device(q->xsk_pool, dma_addr, xdp_desc.len);
        ... // Descriptor is posted to the TX ring
        ...
        q->buffer[ring->head].type = ONIC_TX_XSK;
        ... // Other metadata accounting
        onic_ring_increment_head(ring);
    }
    if (trasmitted) xsk_tx_release(q->xsk_pool);
    onic_set_tx_head(priv->hw.qdma, q->qid, ring->head);
    return trasmitted;
}
```

Listing 2.9: Transmission routine called from the NAPI loop

After a packet has been transmitted, the page has to be reclaimed. Since the XSK knows exactly what pages were in the TX ring and in which order, the reclaiming step consist in calling the `xsk_tx_completed` function, passing as arguments the pool and the number of completion entries seen.

2.4. Additional Features Implemented

Support for newer kernel versions has been added, including necessary compilation adjustments and adaptations to internal API changes, particularly those related to the XDP features flag. These modifications ensure compatibility with the latest kernel updates while maintaining performance and reliability.

Improvements have been made to the ring logic, specifically in the configuration of the QDMA queue. The queue has been adjusted to allow disarming of interrupts, and the return value of `napi_complete_done()` is now used to manage this process more efficiently. Additionally, a fix has been applied to address issues with NAPI return logic, improving overall driver stability and the compliance with the internal kernel API.

The driver's ethtool support has been expanded, providing users with greater control and visibility into network performance. The RSS configuration can now be adjusted, allowing modification of both the table and key. Additionally, the ring size can now be queried and adjusted. Further enhancements include the addition of page pool statistics and packet adapter statistics, which help diagnose packet drop issues.

Finally, a critical bug fix has been implemented to resolve a null pointer exception (NPE) that occurred when removing the driver under sustained load. This fix enhances the robustness of the driver, preventing crashes in high-load scenarios.

These updates collectively improve the driver's performance, configurability, and reliability, ensuring better integration with modern kernel versions and more efficient network operations.

3 | Evaluation

The developing, testing and evaluation was done on the OCTFPGA testbed, located in the UMass cluster federated with Cloudlab[36]. The FPGA used were the AMD/Xilinx U280. The CPU used was a dual socket Intel Xeon Gold 6226R @ 2.90 - 3.90 GHz 16 Cores/32 Threads per socket, L1 1 Mib, L2 32 Mib, L3 44 Mib. All of the experiments were done with hyperthreading disabled, IOMMU off, irqbalance off and the CPU governor set to performance.

3.1. TCP Throughput

A significant outcome of this work is the elimination of the OpenNIC driver as a limiting factor in TCP processing. The transition to utilizing the Page Pool facilitates a single TCP flow achieving a throughput of $\sim$ 32 Gbps, compared to the $\sim$ 12Gbps observed with the baseline method, an increment of 160%.

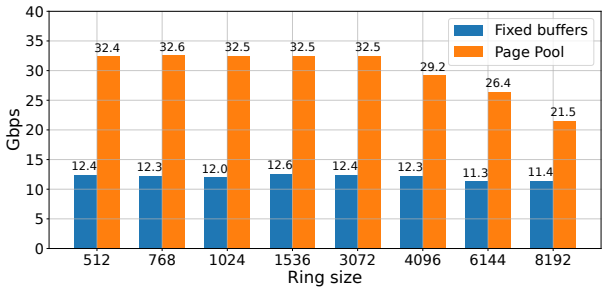


Figure 3.1: Throughput of a TCP single flow. Increasing ring size degrades performance due to higher cache misses.

Figure 3.1 presents the results of the TCP single flow test conducted across different ring sizes at the receiving end. While the blue line remains pretty constant, the page pool line decreases with larger ring sizes. This a known as the *leaky DMA* problem in the literature and its caused by the *Data Direct I/O* feature of Intel's CPU. DDIO is a technology that optimizes I/O performance by allowing network interface cards and other peripherals to directly read from and write to the processor's last-level cache (LLC) instead of main

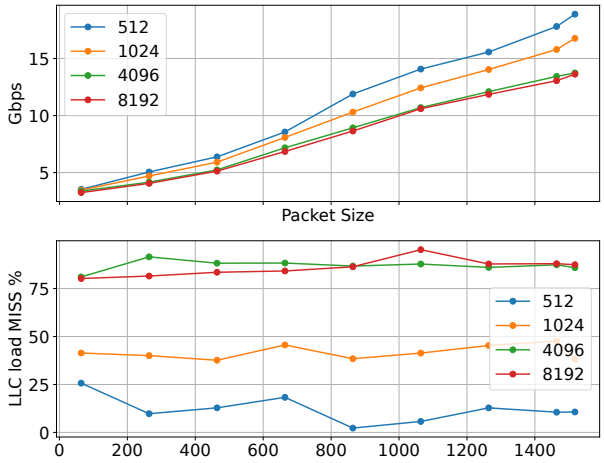


Figure 3.2: Higher LLC load miss correlates with lower throughput

memory. This reduces memory bandwidth usage, lowers latency, and improves throughput for I/O-intensive applications. By bypassing main memory for frequently accessed data, DDIO enhances packet processing efficiency, making it particularly beneficial for high-speed networking workloads. The *leaky DMA* problem arises when large ring sizes (i.e., a high number of RX descriptors) cause excessive cache evictions in DDIO-enabled systems. Since DDIO restricts direct cache access to a limited portion of the last-level cache (LLC), newly arriving packets can overwrite not-yet-processed packets, forcing the processor to fetch them from main memory instead—negating the benefits of DDIO. This issue worsens at high packet rates and large buffer sizes, as packets cycle through the cache too quickly, leading to increased memory bandwidth usage and higher latency. Large ring sizes exacerbate this by allowing more in-flight packets, increasing the likelihood that useful data will be evicted before processing, ultimately reducing network performance at multi-hundred-gigabit speeds [38]. For a NIC that lacks hardware Generic Receive Offload (GRO) and jumbo frame support, like the "bare" OpenNIC shell, the maximum achievable throughput for a single TCP flow is constrained to approximately 30 Gbps due to CPU and memory bottlenecks. Without GRO, packet coalescing must be handled in software, increasing per-packet processing overhead and leading to frequent context switches and interrupt handling. Additionally, the inefficiencies of the Linux network stack at high speeds result in significant CPU utilization, particularly for TCP/IP processing and memory copies. Studies show that while a single core can achieve up to 42 Gbps with optimizations such as GRO and jumbo frames, disabling these features leads to substantial performance degradation [35].

To actually correlate higher cache misses with a decrease in throughput, meant as the amount of packets processed by the receiver, a simple experiment in which a single con-

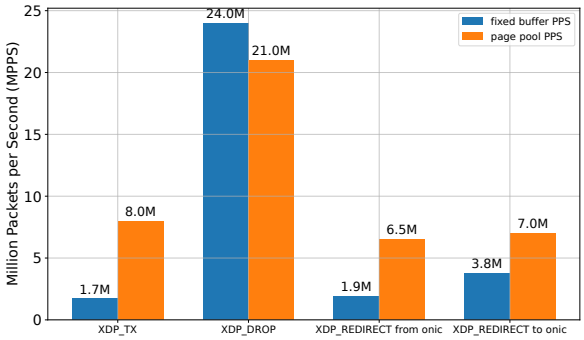


Figure 3.3: Variations of MPPS for the various XDP verdicts.

stant flow of UDP packets was sent, while the receiver had no open sockets. The packets experienced a minimal amount of processing, leading to a minimal amount of memory access per packets. Figure 3.2 shows how the amount of LLC load misses steadily increase along with ring sizes and how it translates in less throughput being processed by the host. It also shows that when the ring size gets bigger than the portion of LLC used by DDIO, which is 10% [38], the cache misses saturates and around 80%.

3.2. XDP performance

This section examines packet processing performance under both single-flow and multi-flow scenarios, leveraging an eight-core system for multi-flow tests and a single core for single-flow evaluations. Packet generation is performed using Pktgen DPDK, which operates in an open-loop mode to provide a consistent packet injection rate. By analyzing how the system handles traffic under different core allocations, we investigate performance bottlenecks, including the impact of ring sizes, cache contention, and DMA inefficiencies.

The number of available cores is constrained by the QDMA IP, which provides only eight interrupt lines per PCIe physical function. The NAPI subsystem (1.1.5) is designed around a 1:1 mapping between interrupts and NAPI instances, ensuring that each core processes packets independently. In theory, the QDMA IP supports up to 2048 queues and an interrupt aggregation ring, allowing multiple queues to share a single interrupt line. However, due to the NAPI restriction, this setup would require multiple NAPI instances to time-share polling on the same core, potentially leading to increased processing overhead and degraded performance.

Figure 3.3 illustrates the improvements in maximum packets per second (PPS) for 64-byte packets when comparing the fixed buffer approach to the page pool, under a single flow scenario. The `XDP_TX` action exhibits a 370% increase in performance, while

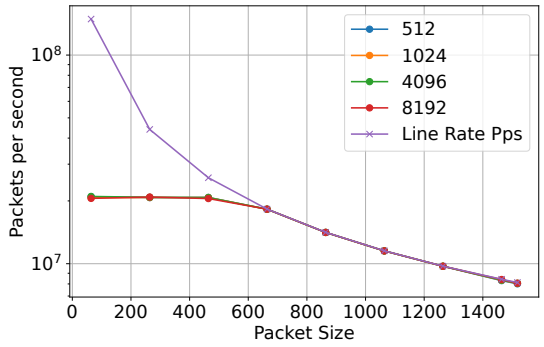
XDP_REDIRECT—initiated in the OpenNIC driver—shows a 240% improvement. Additionally, XDP_REDIRECT targeting ONIC via a Mellanox ConnectX-5 achieves an 80% performance gain. In the latter case, the mlx5 driver already employs the page pool, enabling efficient page recycling and resulting in higher throughput.

Conversely, the performance of XDP_DROP decreases by 12.5%. This decline is attributed to differences in memory handling between the ORDER0 model and the page pool. In the ORDER0 model, the drop verdict is executed by simply skipping the packet and proceeding to the next one, incurring only minor statistical accounting overhead. In contrast, the page pool mechanism requires placing the page into a lockless cache, allocating a new page, and posting its descriptor into the receive descriptor ring. Since the descriptor ring is a coherent DMA mapping, it cannot be cached by the CPU, leading to increased store overhead. Despite this performance penalty, the XDP_DROP action still achieves competitive results. The following in depth performance analysis will cover only the page pool case, as it provide more performance in most cases and it is the one currently implemented in Xilinx's main branch[15].

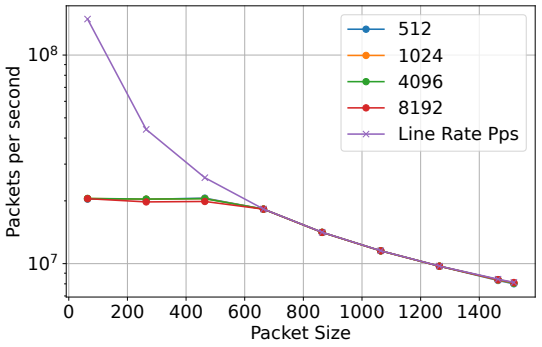
These tests were performed using xdp-bench from the xdp-tools [24] repository. It is a benchmarking utility designed to evaluate the different operational modes of XDP. It serves as a simple yet effective tool for demonstrating various XDP functionalities, including packet dropping, hairpin forwarding (using the XDP_TX return code), and packet redirection via in-kernel redirection mechanisms. Albeit not part of the Linux kernel project it is the de-facto standard tool for benchmarking and regression analysis of XDP implementations of "in-tree" drivers. Statistics accounting is managed using eBPF maps, which introduce a slight overhead to each program. However, a realistic XDP program is likely to include at least one eBPF map for state management, making this additional memory access negligible.

Figure 3.4a and 3.4b show virtually the same performance regardless of the packet write and without performance loss due to leaky DMA. This because XDP_DROP is such a fast operation that never causes packets to be buffered for a long time. The big gap at smaller packet sizes between the theoretical line rate and the actual processed packets is due to intrinsic performance limitations of using a single queue/core for packet processing. Nevertheless for packets bigger than 500 bytes a single queue can drop at line rate.

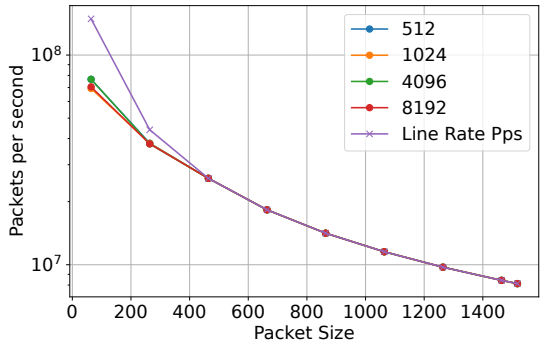
The reduction in drop rate observed between 3.4c and 3.4d in the multi-flow drop-swap-macs scenario for 64 byte packets, 80 Mpps versus 50 Mpps,can be attributed to the leaky DMA problem, which occurs at high packet processing rates when utilizing eight cores. This issue arises because the LLC is shared among all cores, reducing the portion of DDIO



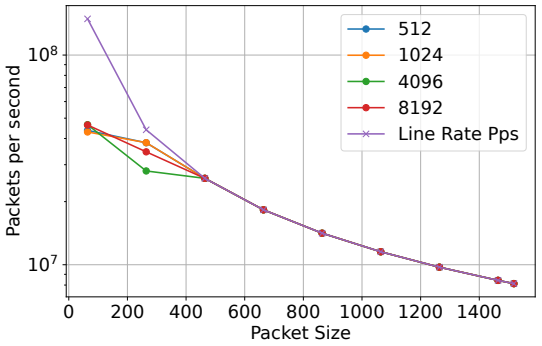
(a) Single flow without reading the packet



(b) Single flow performing a write to the packet



(c) Multi flow without reading the packet



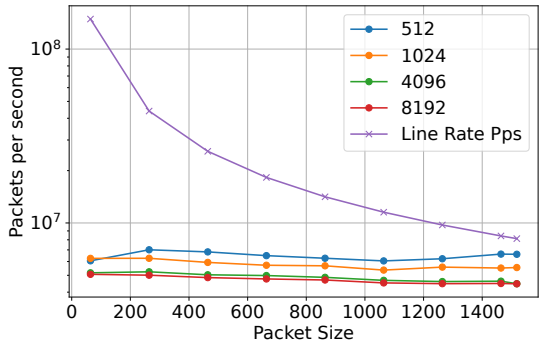
(d) Multi flow performing a write to the packet

Figure 3.4: **XDP_DROP single and multi Flow benchmarks**.

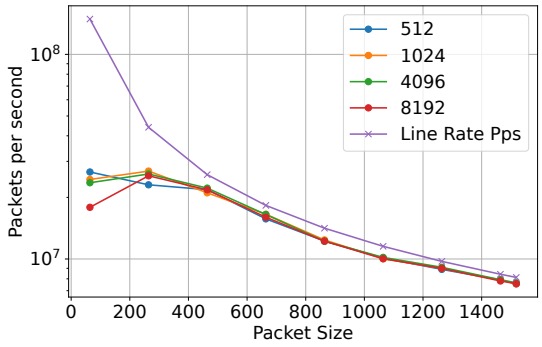
enabled LLC available to each core as the number of processing cores increases.

Single queue `XDP_TX` performance, Figure 3.5a, are limited to 6-8 Mpps due to the overhead of placing the descriptor in the TX queue, perform a DMA sync and later reclaiming, all of which happens on the same core of the RX path. It illustrates the leaky DMA problem caused by larger ring sizes. This issue arises because the driver must synchronously place packets in the transmit queue, leading to increased buffering at the receiving queue. The multi flow small-packet performance of `XDP_TX` in OpenNIC is around 28 Mpps, as shown in Figure 3.5b. The system struggles to handle the high packet arrival frequency, even in full kernel systems like DPDK, the QDMA module encounters difficulties in achieving line rate at small packet sizes [65]. A potential improvement is to configure dedicated TX queues specifically for `XDP_TX`, thereby avoiding the locking overhead described in previous chapters.

The testbed used in this thesis [36] includes a machine equipped with both an Alveo U280 and a Mellanox ConnectX-5. However, the Mellanox card is installed in an x8



(a) Single flow performing a MAC address swap



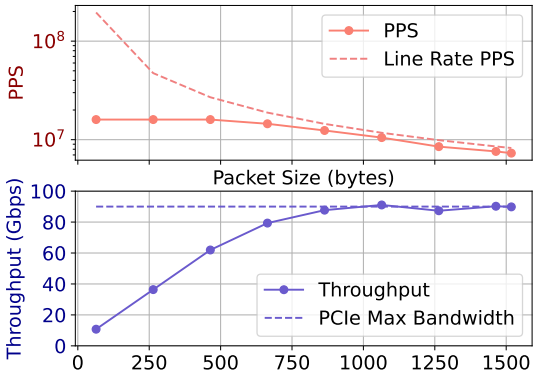
(b) Multi flow performing a MAC address swap

Figure 3.5: **XDP_TX single and multi Flow benchmarks**.

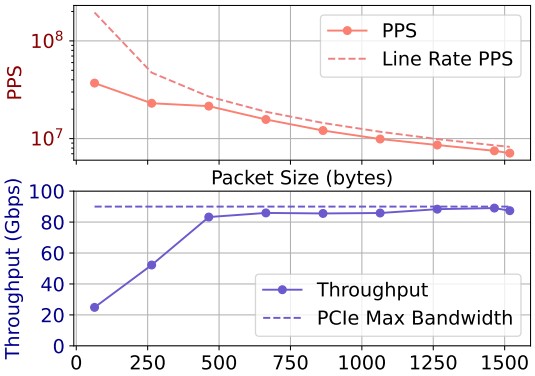
PCIe Gen 3 x16 slot, meaning that while the slot physically supports 16 lanes, only 8 are utilized for data transmission. The maximum theoretical bandwidth of an x8 PCIe Gen 3 slot is approximately 62 Gbit/s, which is further reduced to around 50 Gbit/s when accounting for PCIe protocol overhead [57]. This limitation significantly impacts performance, effectively halving the maximum achievable throughput of the ConnectX-5. To mitigate this issue, I employed two network interface cards—the Mellanox ConnectX-5 and an Intel XL710 40GbE—to simultaneously distribute traffic onto the OpenNIC.

In Figure 3.6, the blue dotted line at 90 Gbps represents the bottleneck caused by the misconfigured mlx5, which contributes approximately 50 Gbps, combined with the Intel card operating at a maximum of 40 Gbps. Figures 3.6b and 3.6c illustrate how OpenNIC's limitation to 8 queues can lead to performance degradation due to excessive queue contention. Both the ConnectX-5 and the XL710 support 64 interrupt lines and, consequently, 64 queues. In order to showcase the best performance the receive ring size was set to 512, and the transmit ring size to 8192.

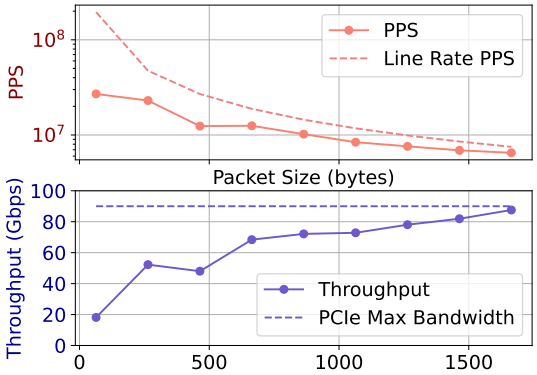
Figure 3.6b shows the redirection performance when using 8 flows per card, which minimizes contention among kernel threads transmitting data. This configuration achieves a maximum rate of 37 Mpps, reaching throughput saturation at a packet size of 500 bytes. In contrast, Figure 3.6c presents results for a scenario with a significantly higher number of flows distributed across all 64 interrupt lines. This leads to a larger number of kernel threads running NAPI instances, all redirecting traffic to OpenNIC. However, since access to the 8 TX queues is synchronized using spinlocks, contention among threads increases, further limiting performance. As a result, the maximum rate decreases to 28 Mpps, with throughput saturation occurring only at a packet size of 1518 bytes. Figure 3.6d shows that redirecting outward from OpenNIC to the two cards achieves a maximum packet



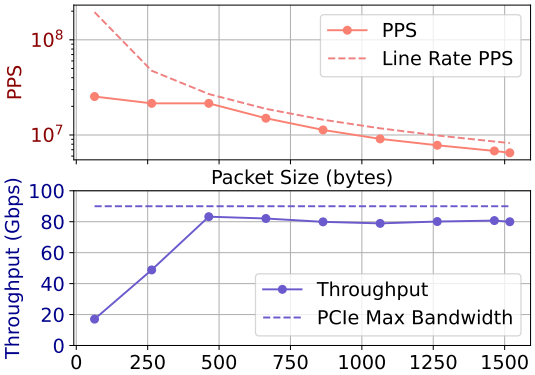
(a) Single flow mlx5 & i40e to onic
.



(b) Small number of flows mlx5 & i40e to onic.



(c) Huge number of flow mlx5 & i40e to onic.



(d) Huge number of flow from onic to mlx5 & i40e.

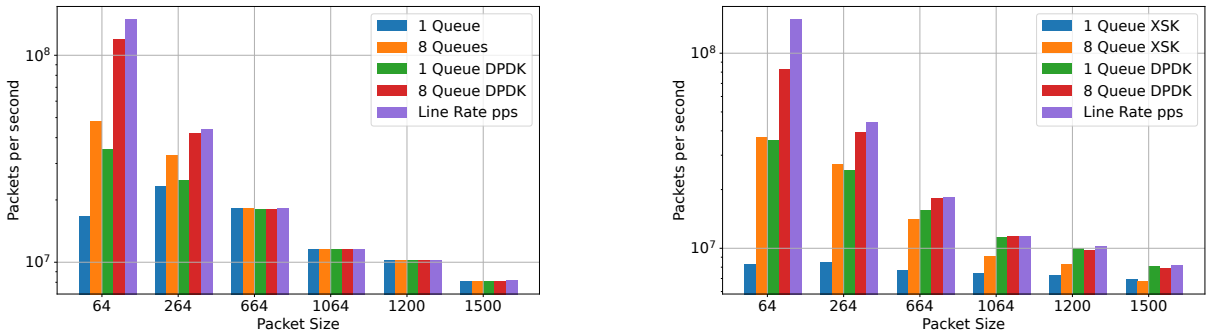
Figure 3.6: **XDP_REDIRECT single and multi flow benchmarks**. The dotted lines show the maximum theoretical pps and the maximum available PCIe bandwidth, capped at 90Gbps due to testbed misconfiguration.

rate of 28Mpps.

3.3. AF_XDP performance

This section presents the performance metrics of the ONIC implementation of XSK. As shown in Figure 3.7a, when using a single queue, the user-space application of XSK achieves a drop rate of 16 Mpps. When utilizing the entire NIC in XSK mode, it reaches 48 Mpps in pure drops. These results align with previous measurements reported in the literature [56] [48].

The plots do not depict variations in ring size because, in XSK, the primary limiting factor is the size of the RX/TX Fill/Completion rings, which are highly application-



(a) Receive Drop

(b) L2 swap forward

Figure 3.7: **XSK benchmark** of two edge cases, a user space application that only drops packets and another that only performs a retransmission, akin to the XDP_TX verdict.

dependent. The tests were conducted using the `af_xdp-example` program, developed as a benchmark by [48]. The UMEM size was set to 4096 frames, and each was sized 2048 frames. Additionally, as shown in Figure 3.7b, the Layer 2 forwarding application achieves 8 Mpps with a single queue and 37 Mpps when utilizing all available queues. Baseline tests using `pktgen-dpdk` as the generator and receiver in the drop case, and `testpmd` as the L2 forwarder, show that while XSK doesn't match the per-queue Mpps performance of a full kernel bypass, it can still reach the throughput of a single DPDK queue by spreading traffic across multiple XSK queues. This suggests that XSK's performance can be improved through parallelism, making queue scalability an important factor. As a result, future OpenNIC development should focus on better aRFS support and optimizing multi-queue handling to improve overall efficiency and performance. This experiments were run in the Run To Completion mode as described in Section 2.3. A comprehensive analysis on the performance characteristic of `AF_XDP` can be found in [58].

4 | Related Works

Open Source FPGA NICs Frameworks The NetFPGA project, started in 2006 [64], has brought 1G, 10G and 100G fully programmable board and open source design to the broader community and a lot of work during the years has been based on these boards. In recent years thanks to the commodification of FPGA boards new platforms have emerged, apart from OpenNIC, like Corundum [40, 41], RecoNIC [68] and ESnet [9]. All of the aforementioned have either a basic network driver or base their driver on top of the OpenNIC one.

Industrial use of SmartNICs Microsoft Azure uses custom and proprietary SmartNICs with FPGA in them for implementing their virtual switch that handles VM-VM communication. [39]. Alibaba Cloud has a similar accelerators called Cloud Infrastructure Processing Unit (CIPU) [7] on which they've build an hardware accelerated fast path for their Apsara vSwitch [50].

Compiling for the hardware Hardware offloads can be either written in "pure" HDL, SystemVerilog or VHDL, or can be compiled from languages and frameworks that are usually used for programming network functions. There exists compilers from eBPF/XDP to HDL like Nanotube [25] and eHDL [59]. hXDP [34] runs the eBPF ISA on custom designed processors running in FPGAs. AMD/Xilinx authored HLS compiler that takes as input P4 programs, called Vitis Networking P4 [23]. VitisNetP4 is also used in ESnet SmartNIC for generating offloads [9].

eBPF/XDP use cases A considerable amount of research has examined the use of eBPF for in-kernel packet processing. The introduction of XDP [46], in particular, has facilitated high-performance implementations of load balancers [19, 22], key-value stores [29, 43, 49], distributed protocols [69, 70], and DDoS mitigation systems [44, 53], automatic detection for offloads has [62, 63], among others. Additionally, eBPF has been leveraged to modify kernel behavior independently of application logic. Examples include adaptive improvements to the TCP/IP stack [67], optimized traffic filtering [51], fine-grained congestion control [45]. While these applications demonstrate the flexibility of

eBPF, substantial research has also identified challenges in developing and maintaining complex network services [27, 30, 52, 54] as well as performance trade-offs when compared to kernel-bypass techniques and hardware offloading solutions [5, 58].

Recent Innovations in the Kernel The Page Pool has recently been updated to work with a new abstract memory type, *netmem*[18] , which serves as an opaque reference to memory that may reside in kernel virtual memory, a userspace-mapped page [11], or an external device such as a GPU [55]. This enhancement is set to become a key component of the next generation of zero-copy network stacks, where headers and metadata are processed by the CPU, while payloads are directly transferred via DMA to their target buffer, regardless of its location, reducing memory copy overhead and lowering the pressure on the host interconnects.

5 | Conclusions and future developments

This work has contributed to OpenNIC, an open-source hardware offload framework for networking applications, by enhancing network stack programmability through the integration of native support for the eXpress Data Path (XDP) and its zero-copy path implementation. The introduction of the Page Pool to the driver has significantly improved performance, achieving up to a 200% increase in single-flow TCP throughput, while also future-proofing the driver codebase for forthcoming enhancements in the kernel memory subsystem. `XDP_DROP` can achieve 50-80Mpps depending on the pattern access to the packets, `XDP_TX` reaches 30Mpps and `XDP_REDIRECT` 37Mpps. XSK benchmarks showed a peak drop rate of 50Mpps and a L2 forwading rate of 40Mpps.

By evaluating XDP performance, I identified limitations in the current implementation of the OpenNIC shell, primarily the limited number of interrupt lines. This constraint, combined with restrictions in the NAPI kernel subsystem, limits the number of CPU cores that can concurrently process incoming packets. Additionally, since the number of receive and transmit queues typically match, transmit performance is also constrained in terms of horizontal scalability.

5.1. Future Work

XDP and AF_XDP Now that zero-copy sockets are supported in OpenNIC, developers can design hardware offloads within the OpenNIC shell, enabling more efficient packet processing. By leveraging these offloads in combination with custom stack processing in `AF_XDP`, developers can minimize CPU overhead, reduce memory copies, and improve packet throughput. This approach allows for high-performance networking applications by optimizing data movement between user space and the network interface, making OpenNIC more suitable for low-latency and high-bandwidth workloads.

Packet Split Starting from Linux v6.9, Page Pool internals are implemented using the newly introduced *netmem* abstraction. This abstraction provides a unified reference type for memory residing in kernel virtual memory, user space, and even hardware devices such as GPUs and accelerators. The OpenNIC shell currently lacks the necessary hardware features to support this paradigm. However, by integrating custom logic and modifying the QDMA layer, it would be possible to implement header-payload split and accelerated Receive Flow Steering (aRFS), both of which are essential for enabling this new zero-copy approach.

Bibliography

[1] AF_xdp - eBPF docs, . URL https://docs.ebpf.io/linux/concepts/af_xdp/.

[2] aya-rs/aya, . URL https://github.com/aya-rs/aya. original-date: 2021-01-20T07:31:17Z.

[3] Bringing eBPF and cilium to google kubernetes engine, . URL https://cloud.google.com/blog/products/containers-kubernetes/bringing-ebpf-and-cilium-to-google-kubernetes-engine.

[4] connectx-datasheet-connectx-8-supernic-3231505.pdf, . URL https://nvdam.widen.net/s/xggffqcjls/connectx-datasheet-connectx-8-supernic-3231505.

[5] Demystifying performance of eBPF network applications. 1(1), .

[6] Design: Having XDP programs per RX-queue, . URL https://xdp-project.net/areas/core/xdp_per_rxq01.html.

[7] A detailed explanation about alibaba cloud CIPU, . URL https://www.alibabacloud.com/blog/a-detailed-explanation-about-alibaba-cloud-cipu_599183.

[8] Dynamic DMA mapping guide — the linux kernel documentation, . URL https://docs.kernel.org/core-api/dma-api-howto.html.

[9] esnet/esnet-smartnic-hw, . URL https://github.com/esnet/esnet-smartnic-hw. original-date: 2022-03-10T19:06:01Z.

[10] General design of queues • versal adaptive SoC CPM DMA and bridge mode for PCI express product guide (PG347) • reader • AMD technical information portal, . URL https://docs.amd.com/r/en-US/pg347-cpm-dma-bridge/General-Design-of-Queues?tocId=kW_aqX_cKbxvaezY4BiYxQ.

[11] io_uring zero copy rx — the linux kernel documentation, . URL https://docs.kernel.org/next/networking/iou-zcrx.html.

[12] L4drop: XDP DDoS mitigations, . URL https://blog.cloudflare.com/l4drop-xdp-ebpf-based-ddos-mitigations/.

[13] libbpf/libbpf, . URL https://github.com/libbpf/libbpf. original-date: 2018-10-10T00:24:34Z.

[14] Linux's circular buffer documentation, . URL https://www.kernel.org/doc/Documentation/core-api/circular-buffers.rst.

[15] Merge pull request #63 from marcomole00/xdp-support-pr · xilinx/open-nic-driver@659b6b7, . URL https://github.com/Xilinx/open-nic-driver/commit/659b6b7c1784d780778801fa2dd8b6a8ccc6aebb.

[16] NAPI — the linux kernel documentation, . URL https://www.kernel.org/doc/html//next/networking/napi.html#id2.

[17] net: Reference bpf_redirect_info via task_struct on PREEMPT_rt. · torvalds/linux@401cb7d, . URL https://github.com/torvalds/linux/commit/401cb7dae8130fd34eb84648e02ab4c506df7d5e.

[18] Netmem support for network drivers — the linux kernel documentation, . URL https://www.kernel.org/doc/html/next/networking/netmem.html.

[19] Open-sourcing katran, a scalable network load balancer, . URL https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/.

[20] Program type 'BPF_prog_type_xdp' - eBPF docs, . URL https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_XDP/.

[21] The RCU API tables, 2019 edition [LWN.net], . URL https://lwn.net/Articles/777165/.

[22] Unimog - cloudflare's edge load balancer, . URL https://blog.cloudflare.com/unimog-cloudflares-edge-load-balancer/.

[23] Vitis networking p4, . URL https://www.xilinx.com/products/intellectual-property/ef-di-vitisnetp4.html.

[24] xdp-project/xdp-tools, . URL https://github.com/xdp-project/xdp-tools. original-date: 2019-06-23T18:58:34Z.

[25] Xilinx/nanotube, . URL https://github.com/Xilinx/nanotube. original-date: 2023-03-03T10:15:00Z.

[26] Xilinx/open-nic, . URL `https://github.com/Xilinx/open-nic`. original-date: 2021-03-01T20:35:05Z.

[27] M. Abranches, E. Hunhoff, R. Eswara, O. Michel, and E. Keller. LinuxFP: Transparently accelerating linux networking. In *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*, pages 543–554. IEEE. ISBN 979-8-3503-8605-9. doi: 10.1109/ICDCS60910.2024.00057. URL `https://ieeexplore.ieee.org/document/10630901/`.

[28] S. Agarwal, R. Agarwal, B. Montazeri, M. Moshref, K. Elmeleegy, L. Rizzo, M. A. De Kruijf, G. Kumar, S. Ratnasamy, D. Culler, and A. Vahdat. Understanding host interconnect congestion. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, pages 198–204. ACM. ISBN 978-1-4503-9899-2. doi: 10.1145/3563766.3564110. URL `https://dl.acm.org/doi/10.1145/3563766.3564110`.

[29] Z. Ahmed, M. H. Alizai, and A. A. Syed. InKeV: in-kernel distributed network virtualization for DCN. 46(3):4:1–4:6. ISSN 0146-4833. doi: 10.1145/3243157.3243161. URL `https://dl.acm.org/doi/10.1145/3243157.3243161`.

[30] T. A. Benson, P. Kannan, P. Gupta, B. Madhavan, K. S. Arora, J. Meng, M. Lau, A. Dhamija, R. Krishnamurthy, S. Sundaresan, N. Spring, and Y. Zhang. NetEdit: An orchestration platform for eBPF network functions at scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, pages 721–734. Association for Computing Machinery. ISBN 979-8-4007-0614-1. doi: 10.1145/3651890.3672227. URL `https://dl.acm.org/doi/10.1145/3651890.3672227`.

[31] K. N. Billimoria. *Linux kernel programming part 2 - char device drivers and kernel synchronization: create user-kernel interfaces, work with peripheral I/O, and handle hardware interrupts*. Packt Publishing. ISBN 978-1-80107-951-8 978-1-80107-082-9.

[32] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: programming protocol-independent packet processors. 44(3):87–95, . ISSN 0146-4833. doi: 10.1145/2656877.2656890. URL `https://dl.acm.org/doi/10.1145/2656877.2656890`.

[33] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. .

[34] M. S. Brunella, G. Belocchi, M. Bonola, S. Pontarelli, G. Siracusano, G. Bianchi, A. Cammarano, A. Palumbo, L. Petrucci, and R. Bifulco. {hXDP}: Efficient software

packet processing on {FPGA} {NICs}. pages 973–990. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/brunella.

[35] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 65–77. ACM. ISBN 978-1-4503-8383-7. doi: 10.1145/3452296.3472888. URL https://dl.acm.org/doi/10.1145/3452296.3472888.

[36] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The design and operation of {CloudLab}. pages 1–14. ISBN 978-1-939133-03-8. URL https://www.usenix.org/conference/atc19/presentation/duplyakin.

[37] A. Farshin, L. Rizzo, K. Elmeleegy, and D. Kostić. Overcoming the IOTLB wall for multi-100-gbps linux-based networking. 9:e1385, . ISSN 2376-5992. doi: 10.7717/peerj-cs.1385. URL https://peerj.com/articles/cs-1385.

[38] A. Farshin, A. Roozbeh, G. Q. M. Jr, and D. Kostić. Reexamining direct cache access to optimize {I/O} intensive applications for multi-hundred-gigabit networks. pages 673–689, . ISBN 978-1-939133-14-4. URL https://www.usenix.org/conference/atc20/presentation/farshin.

[39] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, pages 51–64. USENIX Association. ISBN 978-1-931971-43-0.

[40] A. Forencich, A. C. Snoeren, G. Porter, and G. Papen. Corundum: An open-source 100-gbps NIC. In *28th IEEE international symposium on field-programmable custom computing machines*.

[41] J. A. Forencich. System-level considerations for optical switching in data center networks. URL https://escholarship.org/uc/item/3mc9070t.

[42] B. Gbadamosi, L. Leonardi, T. Pulls, T. Høiland-Jørgensen, S. Ferlin-Reiter, S. Sorce, and A. Brunström. The eBPF runtime in the linux kernel. URL https://arxiv.org/abs/2410.00026. Version Number: 2.

[43] Y. Ghigoff, J. Sopena, K. Lazri, A. Blin, and G. Muller. {BMC}: Accelerating memcached using safe in-kernel caching and pre-stack processing. pages 487–501. ISBN 978-1-939133-21-2. URL https://www.usenix.org/conference/nsdi21/presentation/ghigoff.

[44] Gilberto Bertin. XDP in practice: integrating XDP into our DDoS mitigation pipeline. URL https://api.semanticscholar.org/CorpusID:201112296.

[45] J.-T. Hinz, V. Addanki, C. Györgyi, T. Jepsen, and S. Schmid. TCP's third eye: Leveraging eBPF for telemetry-powered congestion control. In *Proceedings of the 1st workshop on EBPF and kernel extensions*, eBPF '23, pages 1–7. Association for Computing Machinery. ISBN 979-8-4007-0293-8. doi: 10.1145/3609021.3609295. URL https://doi.org/10.1145/3609021.3609295. Number of pages: 7 Place: New York, NY, USA.

[46] 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 *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, CoNEXT '18, pages 54–66. Association for Computing Machinery. ISBN 978-1-4503-6080-7. doi: 10.1145/3281411.3281443. URL https://dl.acm.org/doi/10.1145/3281411.3281443.

[47] Jamal Hadi. When NAPI comes to town. URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=7d7c2566978b10a7fe7e6106b1ecdcf702cf61d5.

[48] M. Karlsson and B. Topel. The path to DPDK speeds for AF XDP.

[49] K. Lazri, A. Blin, J. Sopena, and G. Muller. Toward an in-kernel high performance key-value store implementation. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 268–2680. doi: 10.1109/SRDS47363.2019.00038. URL https://ieeexplore.ieee.org/document/9049596. ISSN: 2575-8462.

[50] X. Li, X. Jiang, Y. Yang, L. Chen, Y. Wang, C. Wang, C. Xu, Y. Lv, B. Yang, T. Wu, H. Gao, Z. Chen, Y. Qiao, H. Ding, Y. Dong, H. Yang, J. Song, J. Lu, P. Zhang, C. Wei, Z. Zhang, W. Chen, Q. He, and S. Zhu. Triton: A flexible hardware offloading architecture for accelerating apsara vSwitch in alibaba cloud. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 750–763. ACM. ISBN 979-8-4007-0614-1. doi: 10.1145/3651890.3672224. URL https://dl.acm.org/doi/10.1145/3651890.3672224.

[51] S. Miano, M. Bertrone, F. Risso, M. V. Bernal, Y. Lu, and J. Pi. Securing linux with

a faster and scalable iptables. 49(3):2–17, . ISSN 0146-4833. doi: 10.1145/3371927.3371929. URL https://doi.org/10.1145/3371927.3371929.

[52] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal. Creating complex network services with eBPF: Experience and lessons learned. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–8. IEEE, . ISBN 978-1-5386-7801-5. doi: 10.1109/HPSR.2018.8850758. URL https://ieeexplore.ieee.org/document/8850758/.

[53] S. Miano, R. Doriguzzi-Corin, F. Risso, D. Siracusa, and R. Sommese. Introducing SmartNICs in server-based data plane processing: The DDoS mitigation use case. 7:107161–107170, . ISSN 2169-3536. doi: 10.1109/ACCESS.2019.2933491. URL https://ieeexplore.ieee.org/document/8789414/.

[54] S. Miano, F. Risso, M. V. Bernal, M. Bertrone, and Y. Lu. A framework for eBPF-based network functions in an era of microservices. 18(1):133–151, . ISSN 1932-4537. doi: 10.1109/TNSM.2021.3055676. URL https://ieeexplore.ieee.org/abstract/document/9340283?casa_token=J9eezmpVGa8AAAAA:n16vXEl688x5Ggwx1ybH9rTl32tlq93V2Tk8Q1PaEh4ju6aBI0B7wxzEZ1ke4h6PFKbGiBiu. Conference Name: IEEE Transactions on Network and Service Management.

[55] Mina Almasry, Willem de Bruijn, Eric Dumazet, and Kaiyuan Zhang. Device memory TCP. URL https://netdevconf.info/0x17/sessions/talk/device-memory-tcp.html.

[56] J. Mostafa, S. Chilingaryan, and A. Kopmann. Are kernel drivers ready for accelerated packet processing using AF_xdp? In *2023 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 117–122. doi: 10.1109/NFV-SDN59219.2023.10329590. URL https://ieeexplore.ieee.org/document/10329590. ISSN: 2832-2231.

[57] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore. Understanding PCIe performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 327–341. ACM. ISBN 978-1-4503-5567-4. doi: 10.1145/3230543.3230560. URL https://dl.acm.org/doi/10.1145/3230543.3230560.

[58] F. Parola, R. Procopio, R. Querio, and F. Risso. Comparing user space and in-kernel packet processing for edge data centers. 53(1):14–29. ISSN 0146-4833. doi: 10.1145/3594255.3594257. URL https://dl.acm.org/doi/10.1145/3594255.3594257.

[59] A. Rivitti, R. Bifulco, A. Tulumello, M. Bonola, and S. Pontarelli. eHDL: Turning

eBPF/XDP programs into hardware designs for the NIC. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, pages 208–223. Association for Computing Machinery. ISBN 978-1-4503-9918-0. doi: 10.1145/3582016.3582035. URL https://doi.org/10.1145/3582016.3582035.

[60] Salim, Jamal Hadi and Olsson, Robert and Kuznetsov, Alexey. Beyond softnet.

[61] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM workshop on hot topics in networks*, pages 150–156.

[62] F. Shahinfar, S. Miano, G. Siracusano, R. Bifulco, A. Panda, and G. Antichi. Automatic kernel offload using BPF. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 143–149. ACM, . ISBN 979-8-4007-0195-5. doi: 10.1145/3593856.3595888. URL https://dl.acm.org/doi/10.1145/3593856.3595888.

[63] F. Shahinfar, S. Miano, G. Siracusano, R. Bifulco, A. Panda, and G. Antichi. Disaggregate applications along end-host data-path. In *Proceedings of the on CoNEXT Student Workshop 2023*, pages 27–28. ACM, . ISBN 979-8-4007-0452-9. doi: 10.1145/3630202.3630230. URL https://dl.acm.org/doi/10.1145/3630202.3630230.

[64] G. Watson, N. McKeown, and M. Casado. NetFPGA: A tool for network research and education. In *2nd workshop on architectural research using FPGA platforms (WARFP)*, volume 3. Citeseer.

[65] Xilinx. Queue DMA subsystem for PCI express (PCIe) - performance report. URL https://adaptivesupport.amd.com/s/article/71453?language=en_US.

[66] Z. Xiong and N. Zilberman. Do switches dream of machine learning?: Toward in-network classification. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pages 25–33. ACM. ISBN 978-1-4503-7020-2. doi: 10.1145/3365609.3365864. URL https://dl.acm.org/doi/10.1145/3365609.3365864.

[67] S. A. Zadeh, A. Munir, M. M. Bahnasy, S. Ketabi, and Y. Ganjali. On augmenting TCP/IP stack via eBPF. In *Proceedings of the 1st workshop on EBPF and kernel extensions*, eBPF '23, pages 15–20. Association for Computing Machinery. ISBN 979-8-4007-0293-8. doi: 10.1145/3609021.3609300. URL https://doi.org/10.1145/3609021.3609300. Number of pages: 6 Place: New York, NY, USA.

[68] G. Zhong, A. Kolekar, B. Amornpaisannon, I. Choi, H. Javaid, and M. Baldi. A

primer on RecoNIC: RDMA-enabled compute offloading on SmartNIC. URL http://arxiv.org/abs/2312.06207.

[69] Y. Zhou, Z. Wang, S. Dharanipragada, and M. Yu. Electrode: Accelerating distributed protocols with {eBPF}. pages 1391–1407, . ISBN 978-1-939133-33-5. URL https://www.usenix.org/conference/nsdi23/presentation/zhou.

[70] Y. Zhou, X. Xiang, M. Kiley, S. Dharanipragada, and M. Yu. {DINT}: Fast {In-Kernel} distributed transactions with {eBPF}. pages 401–417, . ISBN 978-1-939133-39-7. URL https://www.usenix.org/conference/nsdi24/presentation/zhou-yang.

Acknowledgements

This work started as a group project for the Advanced Operating Systems course taught by Prof. Vittorio Zaccaria. A portion of the fixed buffer XDP implementation was co-authored by me and Alexandru Gabriel Bradatan. Of that work very few lines of code have survived the page pool migration, the XSK support and the other work done to the driver. I've learned a lot working with him and I wish him the best of luck.

Thanks to my supervisor Prof. Gianni Antichi for all the help, support and guidance.