BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

USING XDP TO ACCELERATE ROUTING IN THE LINUX KERNEL

VYUŽITÍ XDP PRO AKCELERACI SMĚROVÁNÍ V JÁDŘE SYSTÉMU LINUX

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. SAMUEL DOBROŇ**
AUTOR PRÁCE

SUPERVISOR **Ing. MATĚJ GRÉGR, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2025

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

164490

Institut:	Department of Information Systems (DIFS)
Student:	**Dobroň Samuel, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Computer Networks
Title:	**Using XDP to accelerate routing in the Linux kernel**
Category:	Networking
Academic year:	2024/25

Assignment:

1. Get acquainted with eXpress Data Path (XDP) technology for fast packet processing.
2. Study the standard packet forwarding path in the Linux kernel
3. Perform performance testing of the existing xdp_forward prototype for measuring forwarding speed and cpu utilization and identify problems and missing pieces in the prototype.
4. Evaluate the results and fix potential missing issues of the xdp_forward prototype.
5. Discuss possibility for hardware offloading of the xdp_forward prototype.

Literature:

- Bharadwaj, R. (2017). Mastering Linux Kernel development: A kernel developer's reference manual. ISBN: 978-1-78588-613-3.
- Love, R. (2010). Linux Kernel Development (3rd ed.). Addison-Wesley Professional. ISBN: 978-0-672-32946-3
- Gregg, B. (2019). BPF Performance Tools: Linux System and Application Observability (1st. ed.). Addison-Wesley Professional. ISBN: 9780136554820

Requirements for the semestral defence:
1., 2.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Grégr Matěj, Ing., Ph.D.**
Consultant:	Toke Høiland-Jørgensen
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2024
Submission deadline:	21.5.2025
Approval date:	22.10.2024

Abstract

Traditional Linux kernel packet forwarding faces performance constraints at high network speeds. This thesis evaluates eXpress Data Path (XDP) acceleration through `xdp-forward`, comparing its performance against standard kernel forwarding. Missing features were identified, primarily VLAN support and Netfilter integration. As part of this thesis, two VLAN implementation approaches were developed: one using a patched kernel for direct information retrieval using `bpf_fib_lookup`, and another using eBPF maps for unpatched kernel compatibility. Performance tests demonstrate `xdp-forward` increases packet forwarding rates up to four times over conventional kernel paths, with results varying by hardware configuration. This thesis also discusses future work, including XDP queuing mechanisms and hardware offloading possibilities, though current NIC support remains limited.

Abstrakt

Tradičné smerovanie paketov v jadre Linuxu naráža na výkonnostné limity pri vysokých prenosových rýchlostiach. Táto diplomová práca sa zameriava na akceleráciu pomocou eXpress Data Path (XDP) pomocou nástroja `xdp-forward` a porovnáva jeho výkon so štandardným smerovaním v jadre. Počas vývoja boli identifikované chýbajúce funkcionality, najmä podpora VLAN a integrácia s Netfilterom. V rámci práce boli navrhnuteé dva prístupy k implementácii VLAN: jeden využíva upravené jadro pre priamy prístup k chýbajúcim informáciám cez `bpf_fib_lookup`, druhý používa eBPF mapy, čo umožňuje kompatibilitu aj s neupraveným jadrom. Výkonnostné testy ukázali, že `xdp-forward` dokáže zvýšiť rýchlost spracovania paketov až štvornásobne v porovnaní s bežným smerovaním, pričom výsledky závisia od použitého hardvéru. Práca sa venuje aj možnostiam ďalšieho rozvoja, ako sú fronty v rámci XDP či hardvérové zrýchlenie, hoci podpora zo strany sieťových kariet je zatiaľ obmedzená.

Keywords

xdp-forward, routing performance, high-performance networking, Linux kernel, packet forwarding, XDP forwarding, networking stack

Kľúčové slová

xdp-forward, výkon smerovania, vysoko výkonné siete, jadro Linux, smerovanie paketov, XDP smerovanie, sieťový zásobník

Reference

DOBROŇ, Samuel. *Using XDP to accelerate routing in the Linux kernel.* Brno, 2025. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Matěj Grégr, Ph.D.

Rozšírený abstrakt

Táto diplomová práca sa zaoberá akceleráciou smerovania paketov v jadre systému Linux pomocou technológie eXpress Data Path (XDP). Na úvod sa rozoberá sieťový stack Linuxového jadra – od prijatia paketu cez jeho spracovanie, až po jeho odoslanie. Práca analyzuje kľúčové komponenty jadra vrátane prijímacích a vysielacích front, interakcie s ovládačmi a ostatné mechanizmy pre spracovanie paketov. Následne sa práca v Kapitole 2.6 zameriava na technológiu XDP a jej základ – eBPF (extended Berkeley Packet Filter), analyzujúc ich architektúru a spôsob ich integrácie s jadrom.

Kapitola 3 sa zaoberá identifikáciou chýbajúcich funkcií, ako napríklad podpora VLAN, chýbajúca integrácia s Netfiltrom, absencia pozdržanie paketov na čas a jeho dočasné uloženie, preklad adries a problémy s podporou virtuálnych zariadení. Práca tiež analyzuje obmedzenia vyplývajúce z dizajnu XDP.

Jedno z týchto obmedzení – podpora VLAN pre `xdp-forward` sme implementovali vo dvoch verziách. Prvá vyžaduje upravené jadro, konkrétne správanie pomocnej funkcie `bpf_fib_lookup` a druhý prístup, ktorý využíva eBPF mapy pre uloženie mapovania medzi VLAN ID a čístom rozhrania, to zabezpečuje kompatibilitu so staršími verziami jadra. Vylepšenia čakajú na ich začlenenie do prototypu `xdp-forward`.

Výkonnostná analýza našej VLAN implementácie ukázala mierne zníženie výkonu v porovnaní so základnou verziou `xdp-forward`: približne o 10 % pri IPv4 a o 4 % pri IPv6.

Testovanie výkonu `xdp-forward` v Kapitole 5.4.1 na rôznych sieťových ovládačoch (mlx5_core, ice, bnxt_en a sfc) a architektúrach konzistentne potvrdilo, že XDP dosahuje vyššie rýchlosti smerovania ako štandardné jadro. Najlepšie výsledky dosiahol ovládač Intel ice, kde prototyp `xdp-forward` spracoval až štyrikrát viac paketov s veľkosťou 64 bajtov než klasické smerovanie v jadre.

Zároveň bol pozorovaný výrazný rozdiel vo výkone medzi IPv4 a IPv6. Pri použití `xdp-forward` klesal výkon pre IPv6 o 7 % až 28 % oproti IPv4, zatiaľ čo pri klasickom smerovaní v jadre bol rozdiel menší – v rozmedzí 1 % až 10 %. Vo všetkých relevantných testoch dosahovalo zaťaženie CPU približne 100 % na jadro.

Pri testovaní smerovania pri použití viacerých jadier sa najskôr objavil problém so škálovaním nad 10 Mpps pre kernel. Analýzou v Prílohe A bol identifikovaný problém, ktorý spočíval v zamykaní výstupnej fronty. Použitím fronty, ktorá nevyužíva zámky sa umožnilo lepšie škálovanie výkonu naprieč viacerými jadrami procesora.

Testy skúmajúce vplyv veľkosti paketov odhalili zaujímavé rozdiely medzi metódami smerovania. Pri smerovaní v jadre boli rýchlosti relatívne stabilné bez ohľadu na veľkosť paketu. Naopak, výkon XDP bol citlivejší na veľkosť paketov – väčšie pakety viedli k nižším počtom spracovaných paketov za sekundu.

Záver práce (Kapitola 7) sa venuje možnostiam ďalšieho rozvoja XDP smerovania. Veľký potenciál predstavuje odsun `xdp-forward` priamo na sieťové karty, ktoré by mohlo priniesť až 20-násobné zvýšenie výkonu. Beh `xdp-forward` priamo na hardvéri je však prolemom kvôli svojej komplexnosti a využívaniu funkcií, ktoré nie sú na hardvéri dostupné, vrátane prístupu k smerovacej tabuľke jadra. Ďalšie zaujímavé oblasti na rozvoj zahŕňajú integráciu nových funkcií jadra – napríklad lepšiu integráciu s Netfiltrom prostredníctvom nových pomocných BPF funkcií.

Táto práca dokazuje, že XDP dokáže výrazne zvýšiť výkon pri smerovaní paketov v Linuxe, pričom identifikuje konkrétne oblasti, kde je priestor na rozšírenie funkcionality.

Using XDP to accelerate routing in the Linux kernel

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Ing. Matěj Grégr Ph.D. The supplementary information was provided by Toke Høiland-Jørgensen. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis. I further declare that all text was written by me, with generative large language models used solely as editing tools for grammar correction, stylistic refinement, and improved articulation of my existing ideas. All intellectual content, research, implementation work, analyses, testing, and conclusions are entirely my original work.

.....
Samuel Dobroň
May 21, 2025

Acknowledgements

In the vast expanse of existence, where infinite possibilities collide and diverge, I find myself at this particular intersection of time and space, completing this thesis. The path that led here – a sequence of seemingly random events, choices, and encounters – emerges from the underlying chaos that governs all things, perhaps inevitable, perhaps merely one branch among countless alternatives. Every conversation, every book encountered, every moment of frustration or insight has shaped not just this work but the lens through which I perceive the academic landscape. *Scio me nihil scire* – the recognition of our fundamental limitations stands as relevant today as ever.

I extend my gratitude to my thesis supervisor, who provided guidance while allowing me the freedom to explore. Similarly, my consultant offered invaluable perspective that shaped this work in subtle but significant ways.

To my family, who have been present at every junction of my journey – your support has been the constant in a universe of variables. And to friends who have engaged in countless discussions, challenging assumptions and offering new perspectives – you have influenced this work in ways neither you nor I can fully measure.

In this moment of completion, I acknowledge the butterfly effect of all interactions that have led to this point. A different word, a different meeting, a different thought – any small perturbation in the chaotic system of existence might have altered everything. That we arrived here, in this particular configuration of knowledge and expression, seems both miraculous and ordinary – the paradox at the heart of chaos itself.

Thank y'all.

Contents

Chapter 1

Introduction

The rising volume of network traffic present significant challenges for packet processing systems. Traditionally, the Linux kernel manages network packet forwarding, but this approach encounters performance limitations as network interface speeds increase. The kernel networking stack incurs substantial per-packet processing overhead through operations such as memory allocation, protocol parsing, and traversal of multiple software layers, resulting in reduced forwarding rates. High-end routers and network appliances typically avoid Linux software routing entirely due to these performance constraints, instead relying on specialized hardware with application-specific integrated circuits for packet forwarding operations. Consequently, network engineering has increasingly moved toward offloading kernel networking functionality to dedicated hardware or to processing layers positioned closer to the hardware interface.

This thesis investigates methods for accelerating packet forwarding in the Linux kernel through the eXpress Data Path (XDP). XDP enables high-performance packet processing by allowing extended Berkeley Packet Filter (eBPF) programs to execute at the earliest possible stage of packet reception—directly after the Network Interface Controller (NIC) receives a packet and before the kernel allocates a socket buffer. The work specifically examines `xdp-forward`, a utility within the `xdp-tools` collection that leverages XDP to perform direct packet forwarding at this early processing stage, bypassing substantial portions of the conventional kernel networking stack. This approach positions `xdp-forward` as an intermediate solution between traditional kernel software routing and pure hardware-based forwarding, potentially offering a cost-effective alternative for scenarios that do not justify dedicated forwarding hardware.

Although `xdp-forward` offers considerable performance benefits, it lacks certain features present in the complete kernel stack. This thesis systematically addresses these limitations through several approaches. Section 3 provides a comprehensive analysis of features currently not implemented in `xdp-forward`. Section 5 presents detailed results of performance testing, measuring forwarding rates and CPU utilization.

As a practical contribution, this work implements support for Virtual Local Area Networks (VLANs), one of the identified missing functionalities. These modifications extend `xdp-forward`'s applicability to more complex network topologies. The implementation has been submitted and is awaiting integration into the upstream `xdp-tools` project. Finally, Section 6 examines possible future development, including the possibility of hardware offloading for `xdp-forward` operations to further enhance performance.

Chapter 2

Linux kernel

Operating system is a set of programs that provide an abstraction layer between hardware and user applications [42, Chapter 1.1]. One of the most critical program that is part of operating system is kernel, which manages system resources, provides hardware abstraction, enforces security policies, and handles interrupts. The kernel schedules processes, manages memory allocation, controls input/output operations, and facilitates interprocess communication. Linux kernel implements a monolithic design where the kernel executes in privileged CPU mode (ring 0), enabling direct hardware access and memory management. User programs operate in unprivileged mode (ring 3) and must request kernel services through system calls when requiring access to hardware resources or performing privileged operations. [42, Chapter 1.7]

2.1 User Space and Kernel Space Separation

In modern operating systems, address spaces of processes are separated by memory virtualization. This also applies to kernel and other user processes. This separation is called kernel and user space.

User-space processes operate within isolated virtual address spaces, preventing interference between applications. Each process maintains its own virtual memory mappings, with the kernel managing the translation to physical memory addresses through page tables. [42, Chapter 1.5]

Access to privileged operations from user space processes occurs through the system call interface, as shown on Figure 2.1. System calls transition CPU from user mode to kernel mode through `syscall` instruction on x86_64 architecture. System calls are implemented as wrapper functions that prepare registers according to the x86_64 System V ABI calling convention. The syscall ID must be present in `rax` register[1]. The parameters are passed through registers in the following order: `rdi`, `rsi`, `rdx`, `r10`, `r8`, `r9`. If more parameters are needed, they are passed on the stack. After the kernel processes the syscall request (e.g. `open()` internally calls `do_sys_open()` kernel function), the return value is stored in `rax` register. System calls calling API is defined by x86_64 System V ABI. [28]

[1] syscall IDs may differ based on CPU architecture and can be found in syscall IDs table. For example, at https://gpages.juszkiewicz.com.pl/syscalls-table/syscalls.html

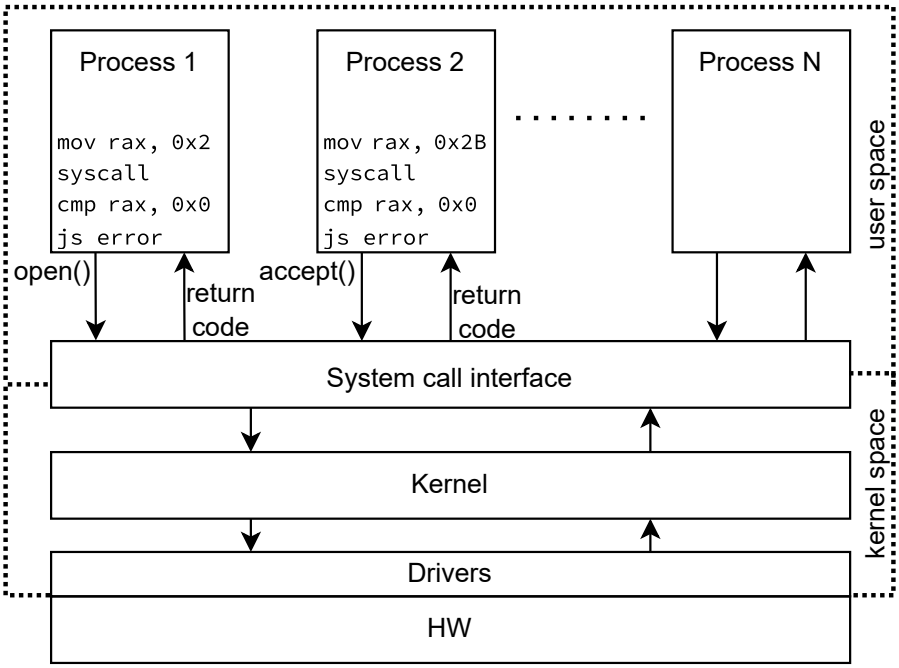

Figure 2.1: User and kernel memory spaces are separated and their communication occurs via well-defined system call interface. This interface provides operations like `open()`, `read()`, or `accept()`.

2.2 Kernel Subsystems

This section is based on [9]. The Linux kernel implements a monolithic architecture where all operating system services execute in kernel space. However, internally the kernel maintains a modular structure divided into distinct subsystems. Each subsystem handles specific system functionality while communicating through well-defined interfaces. This design enables maintainability and allows selective compilation of kernel features based on system requirements.

The core kernel subsystems include:

- Memory Management Subsystem handles virtual memory operations, page allocation, and memory mapping. It implements demand paging, handles page faults, and manages the Translation Lookaside Buffer (TLB).
- Power Management Subsystem controls the system power states, CPU frequency scaling, and device power states through the ACPI interface.
- Process Scheduler manages CPU time allocation between processes, implements scheduling policies, and handles context switching between tasks.
- Sound Subsystem provides audio device abstraction through ALSA (Advanced Linux Sound Architecture), managing sound card drivers and audio streams.
- Networking Subsystem implements network protocols, packet processing, and network device management through the network stack.
- and many others ...

The following sections explain the Networking subsystem in more detail.

2.3 Network Interface Controller

The machine's first contact with Ethernet frame is, when it arrives to Network Interface Controller (NIC) as a series of zeros and ones. The form of these zeros and ones depends on used transmission medium.
However, covering physical layer standards and implementations is outside of this thesis' scope. For anyone interested, we recommend [24]. It guide its' readers through whole Ethernet standard and implementation of NIC in VHDL[2].

So, when the Ethernet frame is received by Network Interface Controller it gets processed and if nothing went wrong it will eventually *somehow* end up in one of RX queues. How it will get to RX queues and what RX or TX queues actually are we will discuss later. Counterpart of RX queues are TX queues where egress packets are positioned until they are transmitted over medium to some other Network Interface Controller [38, Chapter 1].

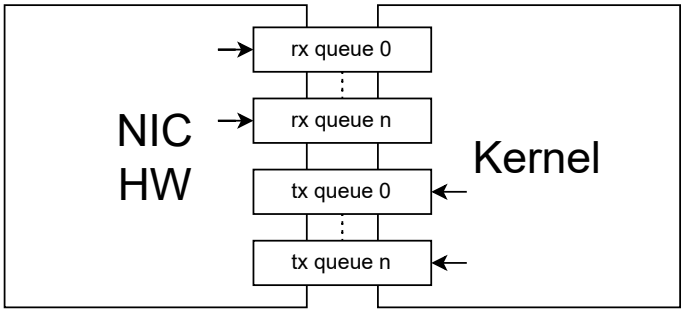

Figure 2.2: Network Interface Card (NIC) and kernel queues illustration showing receive (RX) queues where the NIC hardware writes and kernel reads from, and transmit (TX) queues where the kernel writes and NIC hardware reads from.

2.3.1 Network devices

In the Linux kernel, network devices are abstracted through the `struct net_device` data structure. This structure serves as the primary representation of network interfaces within the kernel, providing a uniform interface regardless of the underlying implementation [38, Chapter 1]. The `struct net_device` contains fields for device properties, statistics, configuration parameters, and most importantly, function pointers that define the operations the device can perform [43].

Network devices are allocated using `alloc_netdev_mqs()`, which manages memory allocation and initialization of the basic device structure. This function requires information about the size of the private data, the device naming, the initialization routine, and the number of transmit and receive queues to configure. After allocation, the device-specific setup function performs hardware-specific initialization [43].

[2]VHDL (VHSIC Hardware Description Language) is a hardware description language used to model and design digital systems, particularly integrated circuits and FPGAs. Unlike traditional and commonly used programming languages that create software instructions for existing hardware, VHDL directly describes the hardware structure and behavior itself, allowing engineers to program the physical circuits of electronic systems [34, Chapter 1].

The `sizeof_priv` parameter allocates driver-specific private data, while `txqs` and `rxqs` specify the number of transmit and receive queues for the device. The `setup` function pointer is called to perform device-specific initialization [43].

After allocation and initialization, the device must be registered with the networking subsystem using `register_netdev()`. This function adds the device to kernel data structures, making it visible and usable by the rest of the system. The registration process performs several important tasks, including assigning a device index, adding the device to network namespace lists, and triggering notification of other subsystems.

The registered `struct net_device` persists in kernel memory even if the driver module is unloaded, ensuring that configuration and statistics remain accessible. The kernel implements a notification chain system through `netdev_chain` to inform other subsystems about device state changes, allowing components to react to network device events without tight coupling. [43]

Linux supports multiple types of network devices that all share this common infrastructure, while implementing different underlying mechanisms for packet processing.

Physical devices

The most basic network device type is physical device, representing actual hardware interfaces typically connected via PCIe that provide connectivity to physical networks. These devices are managed by kernel drivers that enable communication between the networking stack and the hardware.

Physical devices are distinguished by their ability to support hardware offloading features. These offloads move processing tasks from CPU to specialized hardware on the NIC, improving performance and reducing system load [38]. Common hardware offloads include:

- Checksum offload – enables the NIC to calculate and verify checksums for various protocols (IP, TCP, UDP), eliminating CPU overhead for these calculations
- TCP Segmentation Offload (TSO) – allows the kernel to pass large TCP segments to the NIC, which handles the fragmentation into MTU-sized packets
- Generic Segmentation Offload (GSO) – extends segmentation capabilities to other protocols
- Receive Side Scaling (RSS) – distributes incoming traffic across multiple receive queues and CPUs based on hash values computed from packet headers
- Header splitting – separates packet headers from payload data to optimize memory access patterns
- VLAN tag insertion/removal – handles 802.1Q VLAN tags in hardware, where the NIC removes the VLAN tag from incoming packets and sets the corresponding VLAN value in the socket buffer (skb_buff) passed to the kernel

Physical device drivers implement device-specific operations by populating the `net_device_ops` structure with function pointers. These functions include packet transmission (`ndo_start_xmit`), configuration, and statistic collection [38, Appendix A]. The driver may also implement specialized operations like `ndo_xdp_xmit` for XDP program support, which will be discussed later in this thesis.

Stacked devices

Stacked network devices operate as virtual interfaces created on top of other network devices, typically physical interfaces. Unlike physical devices, stacked devices do not directly control hardware but instead process packets before passing them to their underlying devices or after receiving them. These virtual interfaces extend network functionality while leveraging the common `net_device` infrastructure. [38, Appendinx A]

The Linux kernel implements several types of stacked devices:

- VLAN devices – implement IEEE 802.1Q virtual LANs by adding or removing VLAN tags on packets. Created with `ip link add link` *`device`* `name` *`name`* `type vlan id` *`id`*, these devices filter traffic based on VLAN ID. When transmitting, the VLAN device adds the appropriate tag before passing packets to the parent device [23]. The implementation is located in `net/8021q/`.
- Bond interfaces – aggregate multiple network interfaces into a single logical interface, providing increased bandwidth or redundancy [18]. These devices implement various bonding modes such as round-robin, active-backup, and LACP (802.3ad). The bonding driver (`drivers/net/bonding/`) distributes outgoing traffic across slave interfaces according to the selected policy and handles receiving traffic.
- Bridge devices – implement Ethernet bridging (IEEE 802.1D), connecting multiple network segments at the data link layer (L2) [23]. Bridges maintain forwarding databases of MAC addresses to make forwarding decisions. The bridge code (`net/bridge/`) registers hooks in the network stack using the netfilter framework.
- and others.

Stacked devices register with the kernel using the same registration process as physical devices but implement different operations. Each stacked device type registers its own `net_device_ops` structure, with specialized implementations of operations like `ndo_start_xmit` that handle the device's specific packet processing requirements before delegating to the underlying device. [38, Appendinx A]

Virtual devices

Virtual devices exist entirely in software without requiring an underlying physical device. These devices implement the same `net_device` interface but operate solely within the kernel, supporting various virtualization, testing, and tunneling scenarios. [38]

The Linux kernel implements several types of virtualization devices:

- Loopback device – represented by the `lo` interface, provides an internal communication path within the system.
- Veth pairs – virtual Ethernet devices that always come in connected pairs, functioning as a virtual connection between network namespaces. Veth devices transmit packets bidirectionally – packets sent on one end are received on the other.
- TUN/TAP devices – provide a kernel-to-userspace network interface. TAP simulates an Ethernet device and operates at layer 2, while TUN simulates a network layer device and operates at layer 3.
- and others.

2.3.2 NIC driver initialization

In the previous section, we mentioned which network devices are supported by the Linux kernel. In this section, we will focus only on the initialization of physical devices, as the initialization process of other devices is not as important for the topic of this thesis.

Network device initialization is process that establishes the operational state of network interface controllers within the Linux kernel. The initialization sequence begins when the PCI[3] subsystem detects supported network cards, triggering device-specific probe routines [9, Chapter 13]. The initialization process includes several steps: allocation of DMA-capable memory regions for transmitting and receiving descriptor rings, registration of interrupt service routines, configuration of hardware offload capabilities, etc. Before making the device operational, the driver must configure device-specific features such as RSS (Receive Side Scaling) for multi-queue support or its software alternative RPS (Receive Packet Steering). All initialization operations must be completed before device registration, as the network device becomes immediately available for packet processing after successful registration. [14]

Queues allocation

During NIC driver initialization phase[4], the driver allocates a fixed region of memory[5] in RAM for packet processing and configures circular buffers, commonly known as rings or queues. There might be just one or more receive (RX) and transmit (TX) queues. However, most of modern NICs uses multiple RX and TX queues where each queue is usually handled by different CPU[6]. These pre-allocated ring buffers are used by NIC for populating DMA descriptors pointing to memory where the actual packet data are located. [38, 14]

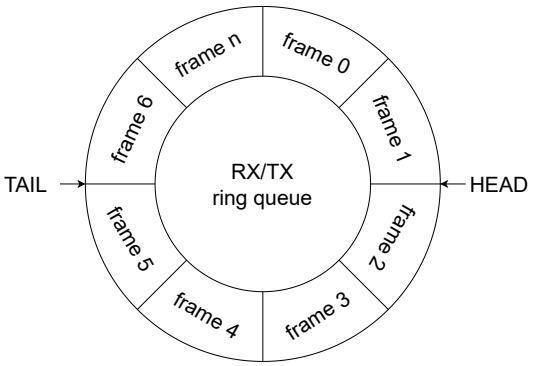

Figure 2.3: Illustration of ring buffer or circular buffer data structure. The reader is reading data from `HEAD` pointer position and writer is writing to `TAIL` position. If writer reaches `HEAD` pointer the buffer is full. In context of RX queue, NIC driver would be writer and CPU would be reader.

[3] Most of the high-end NICs are connected via PCI and so, we are ignoring other methods.

[4] Driver is in most cases, loaded during boot time – during device detection phase. But it may happen during run time as well. For example, when connecting hot plug devices, manual (re)loading of responsible driver, ...

[5] These memory regions require non-swappable, physically contiguous memory blocks [29].

[6] In this thesis, the term CPU refers to a logical or physical CPU core, as this aligns with common Linux kernel terminology.

Registration of handling functions

During the driver initialization phase, the driver must register various callback functions that will be invoked by the networking subsystem during packet processing. These functions are stored in the `struct net_device_ops` *(network device operations)* structure, which serves as an interface between the networking stack and the device driver [38]. It's `ndo_start_xmit` function is called when the networking stack needs to transmit a packet, while packet reception is handled through the `ndo_napi_poll` function within the NAPI (New API) framework. NAPI combines interrupt and polling mechanisms to process packets in batch. [38, 14]

Interrupts

After setting up the operation structure, the driver proceeds with the interrupt handling configuration. Modern NICs typically utilize Message Signaled Interrupts (MSI-X), which enable the mapping of different hardware queues to specific CPU cores. [14, 32]

The interrupt handling mechanism operates as follows [17, Chapter 4] and [14]:

1. When packets arrive, the NIC raises an interrupt
2. The corresponding ISR acknowledges the interrupt and disables further interrupts
3. NAPI polling is scheduled on the appropriate CPU
4. The poll function processes packets up to the specified budget for processing
5. If all packets are processed, interrupts are re-enabled; otherwise, polling continues with refreshed budgets

2.3.3 Spreading packets into queues

In previous sections we explained how receive and transmit queues work, what we did not cover is how the queue for packet is selected. With multiple queues available, the system must determine the appropriate queue for each packet to ensure efficient processing and load distribution. This section relies on [19].

RSS

Receive Side Scaling (RSS) distributes incoming network traffic across multiple receive queues using hardware-computed hash values from packet headers. The hash typically includes IP addresses and port numbers, ensuring that packets from the same flow remain on the same CPU core. This hardware-based approach maintains packet ordering while distributing processing load across cores through a pre-configured indirection table.

RPS

Receive Packet Steering (RPS) provides a software-based alternative to RSS that operates after packets reach RAM. RPS computes flow hashes in software and distributes packets to CPUs based on configurable masks.

2.4 Linux networking stack

In this Linux networking stack walk-through we will primarily focus on IPv4. However, there is little to no difference to IPv6.

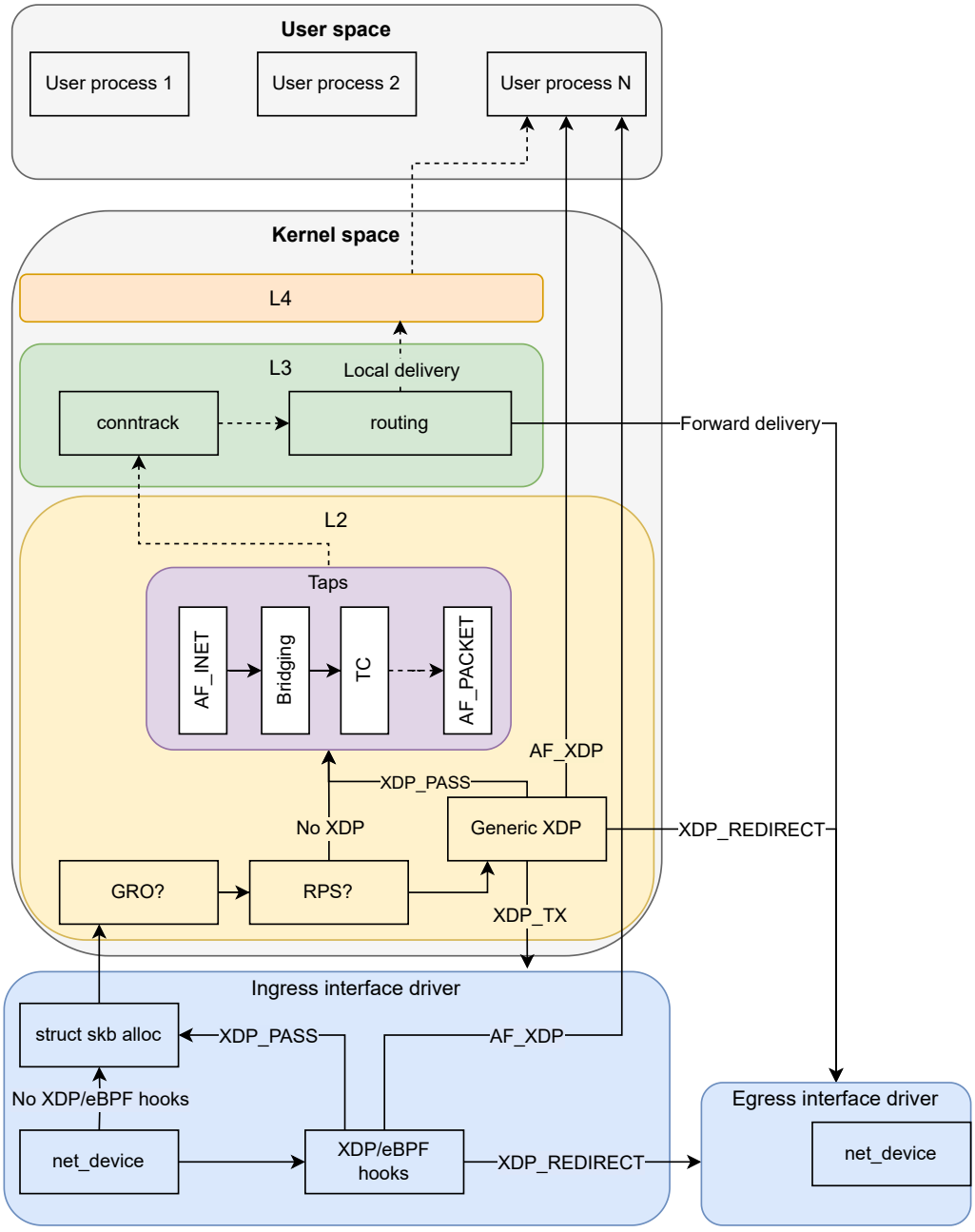

Figure 2.4: Overview of Linux kernel networking stack showing essential components. The diagram illustrates the packet processing paths through different layers, including the XDP/eBPF hooks in the ingress driver, generic XDP processing, and interactions between kernel and user space. Based on [24, 21, 14].

2.4.1 Ingress packet path

The ingress packet path in the Linux networking stack represents the complete journey of a packet from its arrival at the network interface until it reaches its final destination, either a local application or being forwarded to another interface. The complete networking stack architecture is illustrated in Figure 2.4. This section examines each stage of packet processing in detail, following the same sequence a packet encounters as it traverses through the regular networking stack. Understanding this path is essential for network stack optimization and features like XDP that can intercept packets at various stages.

NAPI

The New API (NAPI) provides the interface between network device drivers and the kernel networking stack to receive packets. When packets arrive at the Network Interface Card (NIC), they are placed into RX rings through Direct Memory Access (DMA). The NIC then raises an interrupt to notify the system of new packets. NAPI combines interrupt and polling mechanisms to efficiently transfer these packets from the RX rings to the kernel networking stack. To prevent interrupt storms during high traffic, NAPI employs two control parameters – weight and budget. The weight parameter defines the maximum number of packets that can be processed in one polling cycle per interface, with a typical default value of 64. The budget represents the total number of packets that can be processed across all network interfaces in a single NAPI polling cycle. The default budget is 300 packets. [14]

During low-traffic periods, the interface operates in interrupt mode. When traffic increases beyond a threshold, NAPI switches to polling mode, reducing interrupt overhead. [38, 44]

Passing frames to higher protocols

After GRO and RPS processing, packets are delivered to the appropriate protocol handlers in the networking stack through two main mechanisms – protocol handlers and RX handlers [14]. The core delivery logic is implemented in `__netif_receive_skb_core()`.

```
list_for_each_entry_rcu(ptype, &net_hotdata.ptype_all, list) {
    if (pt_prev)
        ret = deliver_skb(skb, pt_prev, orig_dev);
    pt_prev = ptype;
}
list_for_each_entry_rcu(ptype, &skb->dev->ptype_all, list) {
    if (pt_prev)
        ret = deliver_skb(skb, pt_prev, orig_dev);
    pt_prev = ptype;
}
```

Figure 2.5: Protocol hooks processing. There might be system-wide hooks (`net_hotdata.ptype_all`) and single device only hooks (`skb->dev->ptype_all`). Taken from [44, `net/core/dev.c`, Line 5512].

Protocol handlers provide a way to process packets based on their higher protocol type. Common examples include IPv4 (registered by `ip_packet_type`), IPv6 (`ipv6_packet_type`),

and ARP (`arp_packet_type`). This mechanism is also used by packet capturing tools like Wireshark and libpcap. The kernel maintains two lists of protocol handlers:

1. `ptype_all` – receives all packets regardless of protocol
2. `net_hotdata` – an optimized list for most common protocols, improving performance

For each matching protocol handler, the kernel calls its `.func` member with the packet as parameter. Protocol handlers are registered through `dev_add_pack()` [14]:

```
static struct packet_type my_packet_type = {
   .type = htons(ETH_P_IP),
   .func = my_packet_rcv,
};
dev_add_pack(&my_packet_type);
```

Figure 2.6: An example of `dev_add_pack()` and `struct packet_type` usage.

RX handlers provide a device-level hook mechanism that can intercept packets. This is commonly used by virtual networking infrastructure like bridges. Only one RX handler is allowed per device, registered using `netdev_rx_handler_register()` [14]:

```
static rx_handler_result_t my_rx_handler(struct sk_buff **pskb)
{
   /* Handler logic */
   return RX_HANDLER_PASS;
}
netdev_rx_handler_register(dev, my_rx_handler, NULL);
```

Figure 2.7: An example of `netdev_rx_handler_register()` usage.

netfilter

After packets are processed by protocol handlers, they continue through the network stack where they encounter the netfilter framework. Netfilter is the kernel subsystem that provides the infrastructure for packet filtering, network address translation (NAT), and packet mangling. It operates by defining hook points at various locations in the packet traversal path through the networking stack, with each hook allowing registered functions to examine and potentially modify or drop packets as they pass through. These hook points serve as interception points within the network stack. [38, Chapter 9]

The netfilter defines five primary hook points in the IPv4 and IPv6 processing paths:

- `NF_INET_PRE_ROUTING` – Called before routing decisions, immediately after packet validation
- `NF_INET_LOCAL_IN` – Called when a packet is routed to the local system
- `NF_INET_FORWARD` – Called when a packet is to be forwarded to another host
- `NF_INET_LOCAL_OUT` – Called for outbound packets originating from local processes
- `NF_INET_POST_ROUTING` – Called just before a packet leaves the system

These hook points aren't implemented as separate functions; rather, they are invocation points within key packet processing functions. For example, in the IPv4 implementation, the `ip_rcv()` function calls the `NF_INET_PRE_ROUTING` hook by invoking the `NF_HOOK` macro:

```
int ip_rcv(struct sk_buff *skb, struct net_device *dev,
           struct packet_type *pt, struct net_device *orig_dev)
{
    /* Packet validation and initial processing ... */
    return NF_HOOK(NFPROTO_IPV4, NF_INET_PRE_ROUTING,
                   net, NULL, skb, dev, NULL, ip_rcv_finish);
}
```

Figure 2.8: Netfilter hook invocation in `ip_rcv()`. Taken from [44, `net/ipv4/ip_input.c`].

The `NF_HOOK` macro calls `nf_hook()`, passing protocol family, hook type, network namespace, socket buffer, device pointers, and a callback function to execute if all hooks return `NF_ACCEPT`. Multiple hook functions can register at each hook point, executing in priority order until one returns a non-accept verdict or all complete. The bridge netfilter subsystem extends this with Layer 2 hooks (`NF_BR_*`) that process Ethernet frames before IP processing. Since hooks are executed for every traversing packet, with potentially multiple functions per hook point, they represent significant processing overhead [38, Chapter 9].

conntrack

Following firewall filtering, packets that are allowed to proceed encounter the connection tracking (conntrack) subsystem, which is implemented as part of netfilter as well. Connection tracking is feature of the Linux kernel's networking stack that maintains state information about network connections passing through the system. conntrack enables stateful inspection by associating packets with their respective flows. [38, Chapter 9]

Connection tracking records information about network connections in the kernel's conntrack table. Packets enter the conntrack subsystem through two main entry points:

- `nf_conntrack_in()`: Called at the `PRE_ROUTING` and `LOCAL_OUT` hook points, this function extracts connection information from the packet and either creates a new conntrack entry or updates an existing one. [5]
- `nf_conntrack_confirm()`: Called at the `POST_ROUTING` and `LOCAL_IN` hook points, this function confirms newly created conntrack entries by moving them from the unconfirmed list to the confirmed list. [5]

This connection information is stored in a hash table for efficient lookups, containing state data such as connection status, timeouts, and counters. While enabling essential networking features such as stateful firewalling, conntrack can become a performance bottleneck, particularly for systems handling numerous concurrent connections [35]. The module supports various protocols (TCP, UDP, ICMP, etc.), each with protocol-specific tracking methods implemented through `struct nf_conntrack_l4proto` [38, Chapter 9].

Routing decision

And finally, after all the previous steps, the routing decision needs to be made to decide what are next steps – either delivering packet locally to some of the locally running processes

or, redirecting packet out of (some) networking interface. This will be further discussed in the following section.

2.4.2 Routing using regular Linux networking stack

After processing the packet through the network stack as described in previous sections, the packet reaches either IPv4 or IPv6 layer[7]. Within this layer, the function `ip_rcv` (`ip6_rcv`) performs initial checks on the packet, examining header fields for correctness and verifying checksum. Once these preliminary checks pass, the packet is passed to `ip_rcv_finish_core`, which continues the processing. [44, `net/ipv4/ip_input.c`, Line 456]

As packet traverses through the IP layer processing, it reaches `ip_route_input_slow` function, where the actual routing decision is made. This function determines whether a packet should be delivered locally to a process running on the machine or forwarded to another network interface. After processing broadcast and special case packets, the rest of routing logic is implemented in function `ip_route_input_slow`.

This section examines the routing subsystem in detail, explaining how the Linux kernel makes forwarding decisions based on destination addresses and policy configurations.

Routing tables

The sources for this section were [38, Chapeter 5] and [14, 41] and we discuss the most important routing parts and features. For deeper understanding we advise readers to get through mentioned sources.

The Linux kernel supports multiple routing tables, each with a unique ID, though users typically interact with just the main routing table. These tables map destination networks to next-hop information and are stored internally in an optimized Forwarding Information Base (FIB). This architecture enables advanced scenarios such as policy-based routing. The implementation resides in `net/ipv4/fib_*` files, with `fib_lookup` serving as the primary interface for the networking stack to access routing information.

FIB The Forwarding Information Base (FIB) is the kernel's core data structure for routing information, using a specialized trie-based structure for efficient IP address lookups. This hierarchical organization is optimized through techniques like leaf-pushing and path compression to reduce memory usage and improve cache locality. The kernel maintains separate but algorithmically similar FIB structures for IPv4 and IPv6 in `net/ipv6/route.c` and `net/ipv6/ip6_fib.c`. Beyond packet forwarding, the FIB also supports multipath routing through `struct fib_nh_common` and functions like reverse path filtering and VRF implementation [6].

Lookup process

The routing lookup process in Linux begins when a packet needs to be routed, typically in the context of `ip_route_input_slow` for incoming packets or `ip_route_output_slow` for locally generated packets. The process aims to determine the appropriate path. [38, Chapter 5]

When an incoming packet reaches `ip_route_input_slow`, the kernel first checks if the packet is destined for a local address. This check involves lookup in the local routing table,

[7] Other protocols are outside of this thesis's scope. Therefore, we are omitting them.

which contains routes for the machine's own IP addresses. If the packet is destined for a local address, it is marked for local delivery; otherwise, the routing process continues. [44, `net/ipv4/route.c`, Line 2301]

The actual lookup follows these general steps:

1. The kernel consults the routing policy database (RPDB) to determine which routing table to use. By default, this is the `main` routing table, but policy routing can specify alternative tables. [38, Chapter 6]

2. Within the selected table, the kernel searches the FIB trie structure, traversing it based on the bits of the destination IP address. This search implements the longest prefix match algorithm, where the most specific matching route is selected.

3. The search continues until either a matching route is found or the lookup fails. In case of failure, the kernel may fall back to a default route if one exists.

4. Once a route is found, the kernel extracts the next-hop information and the output device from the routing entry.

The output of the routing lookup process is a routing result structure that contains all necessary information to forward the packet, including next-hop addresses, output interfaces, and any specific handling instructions. [41]

Policy routing

Unlike conventional routing based solely on destination IP, policy routing considers additional parameters, including source address, TOS field, incoming interface, and packet size. The Linux kernel implements this through the Routing Policy Database (RPDB), which contains rules that determine which routing table to use for each packet [38, Chapter 6]. The `fib_rules_lookup` function processes these rules sequentially until finding a match. Rules are represented by `struct fib_rule` and managed via the Netlink interface.

Common applications of policy routing include [41]:

- Source-based routing, where traffic from different source addresses follows different paths
- Multi-homed setups with multiple internet connections, where traffic is directed based on source or destination attributes
- Quality of Service (QoS) implementation, where traffic is routed differently based on TOS or DSCP values
- Traffic isolation in virtualized environments or container setups

Route result

After the routing lookup process is completed successfully, the kernel creates a route result structure that encapsulates all the information needed for packet forwarding. This is represented by `struct rtable` [38, Chapter 5]:

After a route has been selected, the kernel attaches this routing information to the packet's socket buffer (`skb_buff`) by setting its `_skb_refdst` field to reference the destination cache entry. [41]

```
struct rtable {
    struct dst_entry dst; /* Common destination entry fields */
    {
        struct net_device *dev; /* Output device */
        unsigned long expires; /* Cache expiration time */
        /* Function pointers for packet handling */
        int (*input)(struct sk_buff *);
        int (*output)(struct net *net, struct sock *sk, struct sk_buff *skb);
    };
    int   rt_genid; /* Route generation ID~*/
    unsigned int rt_flags; /* Route flags */
    __u16 rt_type; /* Route type (unicast, broadcast, ...) */
    __u8 rt_uses_gateway; /* Whether route uses a~gateway */

    /* Info on~neighbour */
    u8   rt_gw_family;
    union {
        __be32  rt_gw4;
        struct in6_addr rt_gw6;
    };

    /* Path MTU discovery information */
    u32 rt_pmtu;
    /* ... */
};
```

Figure 2.9: Sample of the `rtable` structure from the Linux kernel. Some elements have been omitted. Taken from [44, `include/net/route.h`, Line 56].

2.4.3 Egress packet path

After a packet has been processed through the routing and a route result has been attached to the socket buffer `struct skb_buff`, it enters the egress path. The egress packet path represents the journey a packet takes from the moment it's determined to be forwarded out of the system until it reaches the network hardware for transmission. For IPv4 packets, the egress path typically begins with the `ip_output()` function, which serves as the entry point for the transmission process. Similarly, IPv6 packets enter the egress path through `ip6_output()` [41].

This section examines how packets traverse the various components of the kernel's egress processing pipeline. We are omitting the description of how packets originate from userspace applications and reach the networking stack, as those mechanisms are not directly relevant to the subject of this thesis which focuses on packet forwarding performance.

netfilter

The netfilter framework's involvement in the egress packet path follows a similar pattern to what was described for the ingress path. As packets move through the egress processing pipeline, they encounter predefined hook points where registered netfilter functions can examine and potentially modify or drop them.

For outgoing packets, two primary hook points are relevant [5]:

- `NF_INET_LOCAL_OUT` – This hook is invoked for packets originating from local processes, triggered within the `ip_local_output()` function before routing decisions are finalized.
- `NF_INET_POST_ROUTING` – This hook is the final inspection point before a packet leaves the system, called from `ip_finish_output()` after routing decisions and before neighbor resolution.

These hooks are invoked through the same `NF_HOOK` macro (which ultimately calls the `nf_hook()` function) as described in Section 2.4.1. Like in the ingress path, netfilter hooks in the egress path enable mechanisms such as NAT (particularly SNAT for outgoing connections), stateful packet filtering, and packet mangling operations before packets exit the system. [38, Chapter 9]

Offloading and fragmentation

After packets pass through the netfilter hooks, they enter the `__ip_finish_output()` function which handles packet fragmentation when necessary. This function is responsible for ensuring that the outgoing packets meet the Maximum Transmission Unit (MTU) of the egress interface. The function checks whether the packet size exceeds the MTU of the outgoing interface. If a packet is too large, it must be divided into smaller fragments prior to transmission. [44, `net/ipv4/ip_output.c`, Line 311]

Neighbor address resolution

Packets are then processed by `ip_finish_output2()` function, which handles the neighbor address resolution. This function handles translation of L3 network layer addresses to L2 data link addresses. [44, `net/ipv4/ip_output.c`, Line 230]

For IPv4 packets, this involves searching the Address Resolution Protocol (ARP) cache, while IPv6 packets use the Neighbor Discovery Protocol (NDP) cache. The process follows these general steps:

- Extract the next-hop IP address from the route information attached to the socket buffer
- Search the appropriate neighbor cache (ARP table for IPv4, NDP cache for IPv6) for an entry matching this IP address
- If no entry exists, create a new neighbor entry in an incomplete state
- If an entry exists but is stale or incomplete, initiate address resolution
- If a valid entry exists with a resolved MAC address, use it to build the layer 2 header

When a neighbor's MAC address is not resolved, the system sends an ARP Request (IPv4) or Neighbor Solicitation (IPv6) message and typically queues the packet. The packet remains in this queue until address resolution completes or times out. [38, Chapter 7]

Queuing discipline

After neighbor resolution, packets reach the final stage of the egress path before hardware transmission which is queuing discipline (qdisc). The qdisc subsystem implements traffic control mechanisms in the Linux kernel through software queues that manage packets before they are placed in the hardware transmission queues. [40]

The packets enter the qdisc layer through the `dev_queue_xmit()` function, which serves as the primary interface between the network stack and the device driver layer. This function locates the appropriate qdisc for the outgoing interface and enqueues the packet. [44, `net/core/dev.c`, Line 4340]

The Linux kernel supports various queuing disciplines, each implementing different packet scheduling and traffic shaping strategies. We will mention three of them [40]:

- FIFO (`pfifo` or `bfifo`) – The simplest discipline that processes packets in first-in-first-out order. It offers minimal overhead but provides no traffic management capabilities. The `pfifo` variant limits the queue by packet count, while `bfifo` limits by byte count.
- `fq_codel` (Fair Queuing with Controlled Delay) – A more sophisticated discipline that combines fair queuing with active queue management. It creates separate queues for different flows and applies the CoDel algorithm to manage bufferbloat by monitoring and controlling queue delay.
- **noqueue** – A special discipline that does not actually queue packets. Instead, it attempts to transmit packets immediately or drops them if the device is busy. This discipline minimizes latency at the cost of potential packet loss during congestion.

Once a packet passes through the qdisc layer, it is finally handed to the network device driver via the driver's `ndo_start_xmit()` function. The driver then places the packet in the appropriate hardware transmit queue, typically using DMA operations to transfer the packet data to the NIC's memory [44, `include/linux/netdevice.h`, Line 1031]. After the packet is placed in the hardware queue, the NIC takes over the responsibility for transmitting the packet on the physical medium. The actual transmission occurs asynchronously, with the NIC generating an interrupt upon completion to notify the kernel that the transmit buffer is now available for reuse.

2.5 extended Barkley Packet Filter

Extended Berkeley Packet Filter (eBPF) represents a significant advancement in Linux kernel programmability. While classic BPF was primarily for packet filtering, eBPF has evolved into a general-purpose execution engine within the kernel enabling privileged code execution without requiring kernel modules or system downtime [36].

eBPF operates through a restricted instruction set architecture, with programs either interpreted or compiled to native code via just-in-time compilation. Each program undergoes rigorous verification before execution to ensure system stability and security. Programs can attach to various kernel events including network events, system calls, function entry/exit points, and tracepoints, with access to event context data, state maintenance through maps, and interactions through helper functions [36, 13].

Primary use cases for eBPF include:

- Network programming and packet processing
- Security monitoring and enforcement
- Performance analysis and troubleshooting
- System tracing and observability

The subsequent chapters examine the key components of the eBPF architecture in detail.

2.5.1 eBPF Virtual Machine

The eBPF virtual machine operates within kernel space as a constrained execution environment for verified bytecode instructions. Unlike traditional VMs, it runs within an existing kernel context, maintaining isolation through strict memory access controls while enabling access to kernel functionality via helper functions [36, Chapter 3].

The VM implements a register-based architecture with eleven 64-bit registers: $R0$ for return values, $R1$-$R5$ for argument passing, $R6$-$R9$ as general-purpose registers, and $R10$ as a read-only frame pointer. This architecture mirrors modern CPUs for efficient native code translation [36, Chapter 3].

```
struct bpf_insn {
    __u8 code; /* opcode */
    __u8 dst_reg:4; /* destination register */
    __u8 src_reg:4; /* source register */
    __s16 off; /* signed offset */
    __s32 imm; /* signed immediate constant */
};
```

Figure 2.10: Structure describing a single eBPF instruction in its 64-bit format. Taken from [44, `include/uapi/linux/bpf.h`, Line 77].

Instructions are encoded in either 64 or 128 bits depending on operation type. The opcode field defines operations including arithmetic (`ADD`, `SUB`), bitwise operations, load/store, and control flow instructions [13]. 128-bit format accommodates operations requiring larger immediates or direct memory references.

Memory access is restricted to the stack (limited to 512 bytes), program context, and eBPF maps, with all accesses validated by the verifier before execution [36].

Comparison to regular kernel modules

The execution model differs significantly from traditional kernel modules. While kernel modules execute as part of the kernel's native code with full privileges, eBPF programs operate within the constraints of the virtual machine. [39]

Their execution occurs in response to specific kernel events, with the virtual machine manages the program's lifecycle. When an event triggers, the corresponding eBPF program is executed and clean-up is performed afterwards. The event-driven nature of execution means that eBPF programs only consume resources when actively processing an event, unlike traditional kernel modules that may maintain continuous state and execution contexts. [36, Chapter 3]

2.5.2 Just-In-Time Compilation

The eBPF uses Just-In-Time (JIT) compilation to transform architecture-independent eBPF bytecode (made by regular compilers such as LLVM) into native machine instructions at program load time. The BPF programs are distributed as bytecode rather than architecture-specific binaries, ensuring compatibility across different CPU architectures while maintaining the highest possible execution performance. [13]

Linux kernel supports JIT compilation on multiple architectures, including x86_64, ARM64, PowerPC, SPARC, and RISC-V. Each architecture implements its own JIT compiler backend to handle architecture-specific optimizations and instruction selection. [13]

```
eBPF Instruction      x86_64 Native
----------------      ---------------
mov64 r1, 42      →   mov $0x2a, %rdi
call foo          →   callq <foo>
```

Figure 2.11: Example of eBPF program compilation to architecture specific instructions[8].

2.5.3 BPF Program Verifier

The BPF verifier performs a static analysis before kernel execution to ensure safety and stability. Since BPF programs execute in kernel context, memory access validation is critical to prevent system compromise [39].

Programs must explicitly verify memory boundaries before accessing any regions, including packet data, maps, and context structures. The verifier ensures that all execution paths include proper boundary checks, rejecting programs that might access memory outside their allocated regions [36, Chapter 6].

```
// struct xdp_md *ctx = received data;
void *data = (void *)(long)ctx->data;
void *data_end = (void *)(long)ctx->data_end;

struct ethhdr *eth = data;
if (data + 1 > data_end)
    return -1;
/* do something with packet */
```

Figure 2.12: Example of memory boundaries checking. The program will pass static analysis done by verifier since program checks memory boundaries. Adapted from [20, `common/parsing_helpers.h`, Line 93].

The code verifies the Ethernet header bounds prior to access. Without these checks, a malformed packet could trigger out-of-bounds memory access in kernel space.

[8] The example inspired by https://blog.trailofbits.com/2022/10/12/solana-jit-compiler-ebpf-arm64/.

```
// struct xdp_md *ctx = received data;
void *data = (void *)(long)ctx->data;
void *data_end = (void *)(long)ctx->data_end;

struct ethhdr *eth = data;
/* do something with packet */
```

Figure 2.13: This program will **not** pass static analysis done by verifier because of missing memory boundaries check. Inspired by [20, `common/parsing_helpers.h`, Line 93].

eBPF programs restrictions

This section is based on [13] and [36, Chapter 6].

The verifier enforces several program restrictions to maintain deterministic execution and prevent possible kernel memory corruption:

- Program flow restrictions:
  - No backwards jumps (with exceptions for bounded loops since kernel 5.3)
  - No loops with variable iteration counts
  - No pointer arithmetic that cannot be validated at verification time
  - No calls to arbitrary kernel functions
  - No modification of context structure fields
  - No access to memory outside verified regions
  - No undefined program states
- Resource limitations:
  - Maximum 1 million instructions in total
  - Stack limited to 512 bytes
  - Maximum 32 or 33[9] tail calls in call chain[10]

2.5.4 Maps

As we mentioned before, BPF programs are executed in virtual machine each time an event triggers them. That implies, we need some other mechanism to keep track of BPF programs state – that is where maps come up. Maps implement kernel-space storage that enables data sharing between BPF programs and between BPF programs and user-space applications. The kernel provides several map types optimized for different use cases. [13]

Map definitions specify parameters such as key size, value size, maximum entries, and map type-specific flags. The storage implementation varies by map type, ranging from arrays to hash tables. Access mechanisms implement atomic operations to ensure data consistency during concurrent access from multiple BPF programs or user-space applications [36, Chapter 4]:

[9] Originally, 32 was meant as a limit. Due to off-by-one error it was 33 and other JIT compilers updated their limit to 33 as well due to compatibility reasons. [12]

[10] Tail calls in eBPF provide a mechanism for jumping from one eBPF program to another, similar to function calls but without returning to the original program – functioning analogously to the `execve()` system call. A program can execute up to 32 consecutive tail calls using a `BPF_MAP_TYPE_PROG_ARRAY` map that holds references to other eBPF programs. [12, 36]

- Array maps:
  - Provide direct indexed access with pre-allocated fixed maximum entries
  - There are also per-CPU array maps that eliminate problem with mutal exclusion when multiple CPUs access the map.
- Hash maps:
  - Key-value storage with dynamic entry allocation.
  - Include LRU variants that automatically evict least recently used entries when the map reaches its capacity.
- Ring buffer and perf buffer maps:
  - Implement circular buffer semantics.

Map access from BPF programs occurs through helper functions [15]:

```
void *bpf_map_lookup_elem(struct bpf_map *map, const void *key)
int bpf_map_update_elem(struct bpf_map *map, const void *key,
                        const void *value, u64 flags)
int bpf_map_delete_elem(struct bpf_map *map, const void *key)
```

Figure 2.14: Selection of eBPF helper functions for map access from [15].

User-space applications access maps through file descriptors obtained by calling the *bpf*() system. This approach is used not only for maintaining the state of the BPF program but also for communication between the user-space application and the kernel space BPF program as well. [13]

2.5.5 Kernel API

The Linux kernel provides several interfaces for integrating the eBPF functionality into network device drivers when a trigger event occures. A key interface is the network device operations structure `net_device_ops`, which includes the `ndo_bpf` function pointer. This operation allows network devices to implement device-specific eBPF program handling. [44, `include/linux/netdevice.h`, Line 1336]

2.5.6 Helper Functions

The kernel provides a set of helper functions that eBPF programs can call to interact with various kernel subsystems. These functions implement interfaces for accessing kernel functionality. The verifier ensures that helper function calls use valid arguments and return value handling.

In context of this thesis, several helper functions might be useful [15]:

- `bpf_fib_lookup()` – Performs route lookup in the kernel's Forward Information Base. This function enables programs to determine the next hop for a packet based on destination address.
- `bpf_redirect()` – Redirects packet to another network interface.

- `bpf_clone_redirect()` – Clones and redirects packet.
- `bpf_skb_load_bytes()` – Reads packet data at specified offset.
- `bpf_l3_csum_replace()` – Updates the layer 3 checksum after packet modification.

2.6 eXpress Data Path

The eXpress Data Path (XDP) is a packet processing technology integrated into the Linux kernel that enables high-performance programmable networking. XDP operates at the earliest possible point in the kernel's network stack – directly after a packet arrives from the network interface card (NIC). This placement allows XDP programs to process or filter packets before the kernel allocates any of its internal data structures such as socket buffers (`skb_buff`). XDP programs are written as extended Berkeley Packet Filter (eBPF) bytecode that runs in a sandboxed environment within the kernel. [21, Chapter 1]

```
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

SEC("xdp")
int xdp_drop_all(struct xdp_md *ctx)
{
        return XDP_DROP;
}
```

Figure 2.15: A simple XDP program that will drop all the incoming packets.

When a network interface receives a packet, before any other networking code is executed, the NIC driver calls the registered XDP program through the `ndo_bpf` function pointer stored in `struct net_device_ops` [44, `include/linux/netdevice.h`, Line 1336]. This program can then examine the raw packet data and decide how to handle it using one of several actions – pass it to the networking stack, drop it, transmit it back out, or redirect it to another interface or CPU. `struct net_device_ops` also includes pointer to the function `ndo_xdp_xmit`, which enables packet transmission directly from XDP programs. [44, `include/linux/netdevice.h`, Line 1340]

As shown in Figure 2.4, XDP operates below most of the kernel's networking layers, avoiding their associated overhead. This kernel integration represents a key difference from kernel-bypass approaches like DPDK that move networking entirely to user space. Although DPDK can achieve marginally higher peak performance, it requires dedicating CPU cores exclusively to packet processing and reimplementing functionality already present in the kernel. XDP provides a balanced approach – high performance when needed, while seamlessly integrating with the rest of the system. [21]

2.6.1 XDP actions

XDP programs have direct access to packet data and headers, allowing inspection and modification of any part of the packet. When processing a packet, a program can alter its contents, including header fields and payload data. However, if any modifications are made to protocol headers, the corresponding checksums must be recalculated using BPF-provided helper functions to ensure the packet remains valid. BPF provides functions such

as `bpf_l3_csum_replace` and `bpf_l4_csum_replace` for this purpose. The action return code determines the subsequent processing path for the packet. [21, Chapter 3]

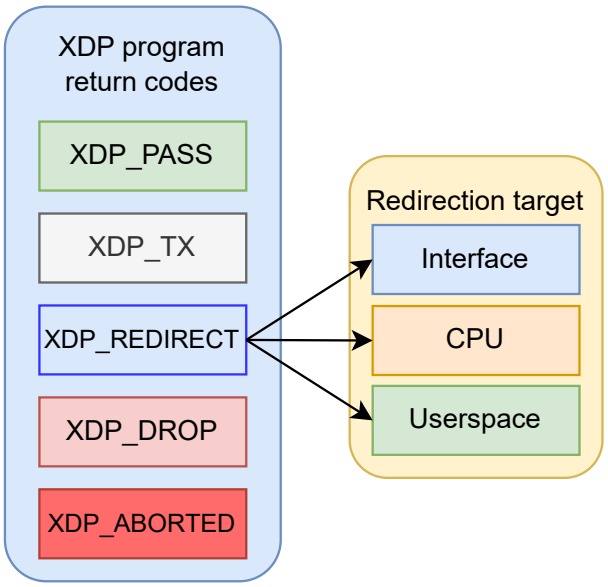

Figure 2.16: The diagram shows the distinct processing paths for each XDP verdict. While `XDP_REDIRECT` requires explicit function calls to handle packet redirection to specified targets (interfaces, CPUs, or userspace), other return codes like `XDP_DROP` and `XDP_PASS` integrate directly into the standard packet processing pipeline without additional execution steps. Inspired by [21, Figure 2].

XDP_PASS

When an XDP program returns `XDP_PASS`, the packet continues through the standard kernel network stack processing path. As shown in Figure 2.4, the packet moves from the „XDP/eBPF hooks“ block to the regular networking stack. At this point, the driver allocates a socket buffer (`skb_buff`) structure and proceeds with standard processing. [36, Chapter 8]

This action is useful when the XDP program needs to inspect or modify packets but wants to maintain compatibility with existing kernel networking functionality. For example, a packet filtering application might use `XDP_PASS` to allow legitimate traffic to reach applications using standard sockets, while dropping suspicious packets using other XDP actions.

XDP_TX

`XDP_TX` action enables packet transmission back through the same interface it was received on, a process known as hairpinning. As illustrated in Figure 2.4, the packet returns directly from the „XDP/eBPF hooks“ block to the originating interface.

When this action is selected, the packet bypasses the kernel and is placed directly into the network interface's transmit (TX) ring buffer for transmission. [36, Chapter 8]

Common applications of `XDP_TX` include:

- Network address translation (NAT) where IP addresses and/or ports require modification
- Protocol translation where packet headers need restructuring
- Echo servers that need to respond on the same interface

XDP_DROP

`XDP_DROP` action provides a mechanism for efficient packet dropping at the earliest possible point in the networking stack. This action is not explicitly shown in Figure 2.4 as the packet processing terminates immediately at the XDP hook. [36, Chapter 8]

When an XDP program returns `XDP_DROP`, the driver deallocates the memory associated with the packet. The memory page containing the packet data is recycled back to the driver's memory pool for reuse with future packets.

This action is commonly employed in:

- DDoS mitigation systems
- Packet filtering applications
- Rate limiting implementations

XDP_ABORTED

`XDP_ABORTED` is an action that, like `XDP_DROP`, terminates packet processing at the XDP hook. However, `XDP_ABORTED` serves a distinct purpose – it indicates an error condition in the XDP program execution rather than an silent packet drop. [36, Chapter 8]

When an XDP program returns `XDP_ABORTED`, the packet is dropped and the memory is deallocated similarly to `XDP_DROP`. However, the action also triggers a trace event. These trace events can be monitored and analyzed using standard Linux tracing tools like `bpftrace` or the `perf`. [20]

XDP_REDIRECT

`XDP_REDIRECT` enables packet redirection to another interface or CPU core. The redirection target is specified by an index parameter, and the actual redirection is performed using either `bpf_redirect` or `bpf_redirect_map` helper functions [25]:

```
int bpf_redirect(u32 ifindex, u64 flags);
int bpf_redirect_map(struct bpf_map *map, u32 key, u64 flags);
```

The redirection is implemented through the target interface's `ndo_xdp_xmit` operation (from `struct net_device_ops`). If the target interface's driver does not implement this operation, packets are silently dropped. The actual packet delivery mechanism is driver-specific. [11]

The `bpf_redirect` function redirects packets based on the network interface index and `bpf_redirect_map` function enables redirection through different map types [25, 11]:

- `BPF_MAP_TYPE_DEVMAP` – redirects packets to other network interfaces. For example:

  ```
  // Redirect packet to interface index 2
  return bpf_redirect_map(&redirect_map, 2, 0);
  ```

- `BPF_MAP_TYPE_CPUMAP` – offloads packet processing to another CPU core. This is useful for load distribution:

```
// Redirect processing to CPU 3
return bpf_redirect_map(&cpu_map, 3, 0);
```

- `BPF_MAP_TYPE_XSKMAP` – redirects packets to AF_XDP sockets for user space processing:

```
// Redirect to AF_XDP socket queue 0
return bpf_redirect_map(&xsk_map, 0, 0);
```

To maintain high performance, XDP redirection operates without packet copying. Once, the packet is transmitted, the page with packet is returned to the originating driver's page pool for reuse. [4]

2.6.2 Generic XDP

Generic XDP provides a software-based implementation of the XDP functionality that operates on socket buffers (`struct skb_buff`) within the kernel networking stack. Unlike native XDP, which processes packets at the driver level before `struct skb_buff` allocation, Generic XDP intercepts packets after they enter the kernel networking stack, specifically in the `__netif_receive_skb_core()` function, as shown in Figure 2.4. [21, Chapter 6]

When a packet arrives, the Generic XDP hook converts the standard kernel `struct skb_buff` to an `struct xdp_buff` before passing it to the XDP program. Based on program's return code, the original `struct skb_buff` with packet is passed upper to the stack or dropped. Transmitting XDP actions transmit the packet on their own and original `struct skb_buff` is dropped:

```
ret2 = do_xdp_generic(device->xdp_prog, &skb);
if (ret2 == XDP_PASS)
    // transmit packet
else
    // drop packet
```

Figure 2.17: Simplified implementation of Generic XDP packet processing in regular Linux kernel networking stack. Adapted from [44, `net/core/dev.c`, Line 5482].

This implementation enables XDP functionality on network interface cards that lack native XDP support in their drivers. The functionality provided by Generic XDP matches that of native XDP implementations – all XDP actions and helper functions are supported. However, the performance characteristics differ significantly. Since packets must traverse part of the kernel networking stack and undergo structure conversion, generic XDP exhibits lower performance compared to native XDP implementations that bypass these steps. This mode can be explicitly requested through the `XDP_FLAGS_SKB_MODE` flag when installing the XDP program, even on network interfaces that support native XDP. [21, Chapter 6]

2.6.3 XDP use cases

XDP's ability to process packets at the lowest possible layer in the networking stack makes it suitable for various high-performance networking applications. The following sections ex-

amine several practical implementations that demonstrate XDP's versatility in production environments.

DDoS Mitigation

DDoS mitigation represents one of the primary applications of XDP, particularly for organizations handling large volumes of network traffic. The early packet interception capabilities of XDP enable efficient filtering of malicious traffic before it reaches the networking stack, reducing system resource consumption during attack scenarios.

Cloudflare uses XDP for DDoS mitigation in their infrastructure as [7] implies. Their implementation processes attack traffic directly at the edge servers rather than relying on dedicated scrubbing centers.

Load Balancing

A notable implementation of load balancer based on XDP is Facebook's Katran[11], an open-source layer 4 load balancer. Katran processes incoming packets at the network edge, using consistent hashing to select destination servers and implementing encapsulation entirely within the XDP program. The system maintains backend server health checks and can dynamically update forwarding rules without interrupting traffic flow.

2.6.4 Routing using XDP forwarding plane

Another use case that XDP can be used for is packet routing. As discussed in Section 2.6.1, XDP programs can use the `XDP_REDIRECT` action to forward packets to different network interfaces after processing. Combined with the `bpf_fib_lookup` helper function (covered in Section 2.5.6), XDP programs can perform routing table lookups directly to determine the appropriate egress interface for each packet.

The basic implementation pattern for an XDP-based router involves several key steps: first, performing a FIB lookup to determine the next hop; second, decreasing the TTL (or hop limit for IPv6); third, rewriting the source and destination MAC addresses based on the lookup result; and finally, calculating any necessary checksum updates. With these operations handled directly at the driver level in XDP program, packets can be forwarded with minimal overhead, bypassing the numerous layers of processing in the standard Linux networking stack.

Existing solutions

There are already several existing solutions that utilize this approach to packet forwarding with XDP. These implementations vary in features, maintenance status, complexity, and design philosophy. Before diving deep into `xdp-forward`, it's worth examining other projects to understand the broader landscape.

XDP router by Honghao Zeng XDP router[12] is an unmaintained[13] proof-of-concept implementation of packet forwarding using XDP. It intercepts IPv4 and IPv6 packets, performs a routing lookup using the `bpf_fib_lookup` helper function, and redirects packets

[11] https://github.com/facebookincubator/katran
[12] https://github.com/Nat-Lab/xdp-router
[13] The last commit was 4 years ago.

directly to the outbound interface using `bpf_redirect`. Packets with other Ethernet types (such as ARP) are passed to the regular kernel networking stack via the `XDP_PASS` return code.

The implementation handles basic packet processing operations, including TTL decrementation and MAC address rewriting based on the next-hop information. When the kernel finds a valid route through its FIB lookup, the XDP program modifies the packet's TTL field (or hop limit for IPv6), updates the Ethernet header with source and destination MAC addresses, and redirects it to the appropriate egress interface.

Although the project claims to support VLAN handling, analysis of the source code reveals significant limitations. The code simply strips the VLAN headers from the incoming packets without preserving VLAN ID information. This contrasts with proper VLAN offload handling (described in Section 2.3.1) where VLAN IDs are preserved in the socket buffer metadata. In addition, the implementation lacks support for forwarding packets to VLAN interfaces. The README documentation includes a misleading statement regarding VLAN interfaces:

> To enable XDP for VLAN interfaces, enable XDP for their master interface. VLAN headers are automatically stripped and the encapsulated IP/IPv6 packets are routed. You may load the XDP executable on VLAN interfaces, but it will be in the „generic“ mode and will not have any significant performance benefits, as the VLAN driver does not have XDP support.

This statement is technically incorrect, as XDP programs cannot be attached to VLAN interfaces at all [2, 8]. The kernel does not support loading XDP programs on virtual devices like VLAN interfaces, even in generic mode.

XDP Proxy by Christian Deacon Another approach to XDP-based packet processing is XDP Proxy[14], a well-maintained project that primarily focuses on implementing a stateless NAT-like proxy system. While this project can perform packet forwarding using XDP, its fundamental design is centered around NAT functionality with source-port mapping similar to traditional tools like `iptables` and `nftables`. This focus on being a proxy rather than a pure forwarding solution introduces significant complexity. The project lacks certain networking features like VLAN support, but this is secondary to the core limitation that it was designed with different goals in mind than efficient packet forwarding. Despite being actively maintained with regular updates, its architecture is more optimized for scenarios like address translation.

xdp-forward The xdp-forward utility is a component of the xdp-tools collection, a set of utilities and libraries maintained by Toke Høiland-Jørgensen, a key contributor to XDP in the Linux kernel. `xdp-forward` is actively maintained with regular updates and releases. The project is distributed as part of `xdp-tools` under the GPL-2.0 license.

The implementation builds upon the original `xdp_fwd` kernel sample written by David Ahern, but has been significantly expanded and integrated with the `libxdp` library to provide better usability and kernel integration. The project is designed to work with modern kernel features and maintains compatibility with current Linux kernel releases.

The utility is implemented using a combination of userspace control code written in C and eBPF programs loaded into the kernel. The architecture follows the separation between

[14] https://github.com/gamemann/XDP-Forwarding

the control plane (userspace) and the data plane (kernel). The userspace component handles configuration, loading of eBPF programs, and management of the forwarding plane, while the eBPF programs executed in kernel context perform the actual packet forwarding.

This thesis was specifically designed to focus on `xdp-forward` as the reference implementation for XDP-based packet forwarding, due to its active maintenance, robust design and integration with the Linux kernel development ecosystem. A detailed analysis of its functionality and performance characteristics will be presented later in Section 3.

Chapter 3

Features Analysis

As established in previous chapters, XDP operates at the earliest possible point in the network processing path, intercepting packets directly within the device driver before entering the Linux networking stack. This positioning, illustrated in Figure 2.4, provides significant performance advantages, but also introduces functional constraints compared to the full Linux networking stack.

This chapter examines the feature limitations of the `xdp-forward` implementation in contrast to standard kernel-based packet forwarding. Due to XDP's placement within the device driver context, the analysis focuses on driver-level features that have direct implications for packet forwarding functionality. Understanding these limitations is critical for determining appropriate deployment scenarios in which XDP-based forwarding offers advantages while maintaining the necessary networking capabilities.

3.1 Firewalling

Packet filtering is an essential component of network management, typically implemented through the Linux netfilter subsystem with userspace interfaces like `nftables` or `iptables`. However, since XDP operates at the driver level, bypassing most of the networking stack, it cannot leverage traditional firewalling mechanisms that operate at later stages.

The netfilter hooks are inserted at specific points in the networking stack's packet traversal path, as shown in Figure 2.4. By the time a packet reaches these hooks (e.g., `NF_INET_PRE_ROUTING`, `NF_INET_LOCAL_IN`), it has already been processed well beyond the XDP layer. This difference presents significant challenges for integrating standard firewalling capabilities with XDP-forward.

Several approaches could theoretically enable firewall functionality within the XDP context:

- Reimplementing firewall machinery in XDP: – This would require a background userspace process to translate, update `nftables` rules, and pass them to corresponding XDP programs. Complete compatibility would require reimplementing the entire netfilter subsystem within XDP. That would duplicate much of the existing kernel functionality and would need to be properly maitained.

- Selective bypass using LPM tries – Using `BPF_MAP_TYPE_LPM_TRIE` to store IP addresses of sources and/or destinations that should be processed by the regular kernel stack rather than `xdp-forward`. This approach would require a userspace pro-

cess to maintain the IP trie and would effectively split traffic into two processing paths, XDP-accelerated and kernel-processed packets. For traffic patterns where only a small networking subset needs filtering, this approach could maintain performance advantages. However, if most packets require filtering, performance would degrade to kernel-stack forwarding with additional XDP overhead.

- New BPF helper for netfilter integration – Creating a new helper function that could invoke netfilter hooks (similar to how `NF_HOOK()` is called in `ip_rcv()` at [44, `net/ipv4/ip_input.c`, Line 569]). This would require constructing a temporary socket buffer structure (`struct skb_buff`) from the raw packet data, which would introduce significant overhead. Each packet would need parsing (to fill up this temporary packet structure) and buffer allocation, largely negating XDP's performance benefits.

Beyond these native approaches, XDP does support loading multiple programs on the same interface, creating a potential fourth option: using a dedicated XDP-based firewall program alongside `xdp-forward`. Tools like `XDP-Firewall`[1] offer this capability, although they typically require manual translation of existing nftables rules.

The current implementation of `xdp-forward` does not include integrated firewalling capabilities. Users requiring both high-performance forwarding and packet filtering must either:

- Accept the performance cost of using kernel-based forwarding with netfilter
- Implement a custom solution combining `xdp-forward` with specialized XDP filtering programs
- Use `xdp-forward` for high-volume traffic patterns where filtering is unnecessary

3.2 Connections tracking

Another mechanism that could benefit from having a BPF helper with netfilter hook integration is connection tracking (conntrack). The Linux kernel implements connection tracking through the conntrack subsystem, which operates via netfilter hooks in the standard networking stack.

Two potential approaches exist for enabling conntrack functionality in XDP:

- Implementing a BPF helper function as mentioned in Section 3.1.
- Implementing a BPF helper function dedicated for access to connection tracking table and implementing packet manipulation operations in XDP programs
- Creating a parallel connection tracking implementation in XDP with a userspace daemon to synchronize state with the kernel's conntrack tables—while feasible, this introduces potential consistency issues and duplicates functionality

Without direct conntrack access, XDP applications requiring stateful processing must either implement limited tracking functionality using BPF maps or pass certain traffic to the regular networking stack.

[1] https://github.com/gamemann/XDP-Firewall

3.3 Address translation

Network address translation (NAT) in the Linux kernel is implemented using connection tracking. Therefore, before adding support for NAT, support for connection tracking is required.

There already is support for network address translation in `xdp-forward` in `flowtable` mode. However, this is a separate BPF program, that reimplements whole network address translation in BPF program. After consulting this, we decided to focus on `fib` mode.

3.4 Packet queuing

Packet queuing plays a critical role in network traffic management, providing mechanisms for buffering, prioritization, and scheduling of packets. While the regular Linux networking stack incorporates multiple queuing mechanisms at different processing stages, XDP's positioning at the earliest point in the packet processing path means it lacks many of these capabilities.

In the standard Linux networking stack, packets encounter several queuing stages as they traverse the system:

- RX/TX ring buffers at the NIC level, which provide initial buffering for incoming and outgoing traffic as discussed in Section 2.3.2
- CPU input queue backlog when RPS (Receive Packet Steering) is enabled, where packets are queued before processing by a target CPU
- Queue discipline (qdisc) layer on the transmit path, which implements sophisticated packet scheduling algorithms described in Section 2.4.3

XDP, however, operates directly in the driver context before packets reach most of these queuing stages. The only queuing mechanism available to XDP is effectively the NIC's hardware ring buffers.

This lack of queueing capability significantly constrains XDP's ability to implement important networking features such as:

- Quality of Service (QoS) policies that require packet prioritization
- Bandwidth shaping to control traffic rates between interfaces of different capacities
- Handling of traffic spikes through controlled buffering
- Fair allocation of resources among multiple flows
- Rate limiting and policing

Several efforts have emerged to add queueing support to XDP. A notable initiative began in 2022 when this thesis consultant Toke Høiland-Jørgensen submitted a Request for Comments (RFC) patch series to the Linux kernel mailing list [3]. This proposed implementation, known as XDP Queuing (XDQ), introduced a new PIFO (Push-In First-Out) map type for storing XDP frames, a dequeue program type for the TX softirq context, and helper functions for queue management.

As described in [3], XDQ enables packet scheduling by allowing packets to be enqueued with assigned priorities, while always dequeuing from the head of the queue. This approach

maintains XDP's performance advantages while adding critical queueing functionality. The performance impact of adding queueing to XDP appears reasonable. According to preliminary testing in [3], queueing adds approximately 50ns of overhead per packet, which translates to about a 20% reduction in packets-per-second throughput compared to standard XDP forwarding. However, this still provides approximately twice the performance of the regular networking stack.

Despite these efforts, XDP queueing support remains an ongoing development effort as of 2025.

3.5 Virtual devices

XDP operates at the physical device driver level, which creates inherent complications for supporting virtual networking devices. Virtual devices represent abstract network interfaces that typically do not have direct hardware access. Therefore, their support in XDP is limited.

The list of virtual devices supported by XDP is significantly limited compared to the standard kernel networking stack. This limitation stems from fundamental architectural differences: virtual devices typically process packets as part of `__netif_receive_skb_core()`, where handler from their subsystem is called. Supporting these virtual interfaces would require either reimplementing their functionality entirely within XDP or modifying kernel interfaces to interact with XDP hooks.

For example, MACsec (IEEE 802.1AE Media Access Control Security) encryption is implemented in the Linux kernel as an `rx_handler` (as discussed in Section 2.4.1). When a packet arrives at a MACsec-enabled interface, the kernel calls the `macsec_handle_frame` function to perform decryption and authentication before passing the packet to higher layers. This processing occurs after the initial driver processing, where XDP hooks are executed. Similarly, on the transmit path, MACsec does not implement the XDP-specific functions required for integration.

Other virtual devices face similar constraints. According to discussions in [2], supporting virtual devices like VLANs and bonded interfaces in XDP would require either:

- Complete reimplementation of the virtual device functionality within XDP programs
- Substantial modifications to kernel device drivers to support XDP at appropriate points in the processing path

David Ahern attempted to implement support for VLANs and bonded interfaces in XDP as documented in [2]. Despite his effort, the implementation remained unfinished, with even the partial patch requiring over 600 lines of code changes to kernel internals.

The following subsections examine two common virtual device types, VLANs and bonded interfaces, in more detail.

3.5.1 Bonds

Link aggregation, implemented in Linux through bonded interfaces, presents another important virtual device type for networking infrastructure. Bond interfaces combine multiple physical network interfaces into a single logical interface, providing increased bandwidth and redundancy. The Linux kernel's bonding driver (`drivers/net/bonding`) supports multiple operational modes including round-robin, active-backup, 802.3ad (LACP), and load-balancing configurations.

Unlike VLAN interfaces, bond interfaces have gained XDP support thanks to Jussi Maki's contribution [26]. This implementation differs significantly from how bonding operates in the standard networking stack.

On the receive path, packets arrive directly at the physical (slave) interfaces and are processed by XDP programs as if no bonding were configured. This approach works because bond interfaces, do not require packet manipulation for received traffic—the bond abstraction exists primarily at the device management level rather than the packet processing level.

The more complex aspects of the implementation occur on the transmit path, where the patch reimplements the slave selection logic from the bond driver directly within the XDP context. This includes support for:

- Round-robin mode, which distributes packets sequentially across available slaves
- Active-backup mode, which maintains a primary interface with failover capabilities
- 802.3ad mode, which implements the IEEE 802.3ad Link Aggregation Control Protocol
- XOR mode, which assigns flows to specific slaves based on a hash of packet headers

After selecting the appropriate slave interface, the implementation invokes that physical interface's `ndo_xdp_xmit` function to transmit the packet [44, `bonding/bond_main.c`, Line 5512].

The bond implementation demonstrates a key pattern for supporting virtual devices in XDP. Rather than attempting to integrate with the kernel's existing virtual device infrastructure (which operates above the XDP layer), the approach reimplements the minimal necessary functionality directly within the XDP context.

3.5.2 VLANs

Virtual LANs (VLANs) represent one of the most common virtual network interfaces in production environments. The Linux kernel implements VLANs through the `8021q` module, which creates virtual interfaces that handle IEEE 802.1Q tagged frames. When packets arrive with VLAN tags, the kernel processes them through specialized handling routines that strip the tags before passing the inner packet to the appropriate virtual interface.

VLANs is a good fit for potential XDP integration compared to other virtual devices for several reasons:

- The VLAN header has a fixed format and position within the Ethernet frame
- VLAN processing primarily involves header manipulation (adding/removing tags)
- The mapping between VLAN IDs and virtual interfaces follows a direct pattern

VLAN support represents a logical first step in extending `xdp-forward` functionality. Section 4 explores practical approaches for implementing VLAN support in `xdp-forward`.

Chapter 4

Extending `xdp-forward` functionality

The previous chapter identified several features missing in `xdp-forward`. While some of these features would be too complex to implement (like netfilter integration) or are already being worked on by others (like XDP queuing [3]), support for virtual devices represents a practical area for improvement within this thesis scope. VLAN support was selected as the implementation target for several reasons. First, VLANs are commonly used in many network environments, making this enhancement practically useful. Second, while bond interfaces already have XDP support through Jussi Maki's work [26], VLANs remain unsupported.

4.1 Design

Before the actual implementation, it is necessary to understand how VLAN tagging works at the packet level and examine possible implementation approaches within the XDP framework.

4.1.1 VLANs

Virtual LANs (VLANs) allow to logically segment a single physical network into multiple broadcast domains. In Linux, VLANs are implemented as virtual network interfaces that operate on top of physical interfaces. As shown in Figure 4.1, a VLAN Ethernet frame differs from a standard Ethernet frame by the insertion of a 4-byte VLAN tag between the source MAC address and the EtherType field. This tag consists of a 2-byte Tag Protocol Identifier (TPID) set to `0x8100` for the IEEE 802.1Q standard, followed by a 2-byte Tag Control Information (TCI) field that contains the VLAN ID and priority information.

In the Linux kernel, when a VLAN-tagged packet arrives, the function `__netif_receive_skb` calls `vlan_do_receive`. If a tag is found, it gets removed, updates the `struct skb_buff` with VLAN metadata, and continues processing as usual. Conversely, when transmitting packets through a VLAN interface, the `vlan_dev_hard_start_xmit` function is called. This function adds a VLAN tag to the packet before passing it to the underlying physical interface. The VLAN ID used for tagging is determined by the configuration of the VLAN interface itself.

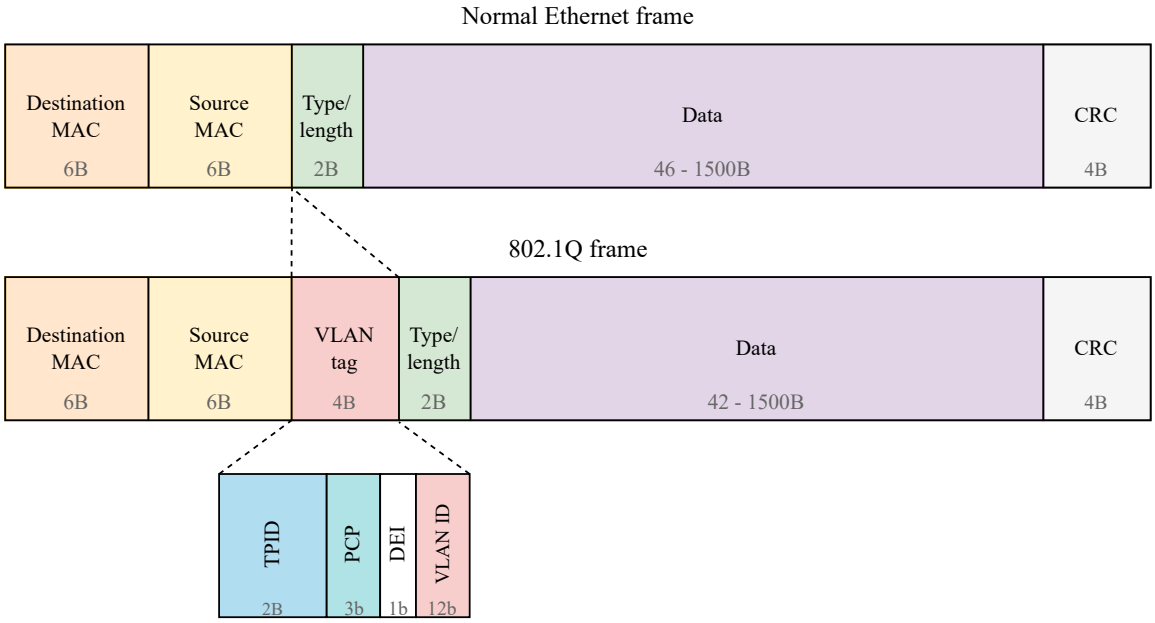

Figure 4.1: Comparison of standard Ethernet frame and IEEE 802.1Q VLAN-tagged frame. The VLAN-tagged frame inserts a 4-byte VLAN tag between the source MAC address and EtherType field. To accommodate this 4-byte tag without exceeding the maximum Ethernet frame size, the space available for data payload is reduced by 4 bytes.

4.1.2 Approach

The main issue with implementing VLAN support for `xdp-forward` is that XDP programs cannot be attached to VLAN interfaces [8]. Therefore, the underlying physical interface's XDP program must handle all VLAN processing.

When processing packets, the XDP program needs to handle several VLAN scenarios:

- Replace existing VLAN tag when forwarding between different VLAN networks
- Remove VLAN tag when forwarding from VLAN to untagged network
- Add VLAN tag when forwarding from untagged to VLAN network

The structure returned by `bpf_fib_lookup` function has fields for VLAN information:

```
struct { /* output */
    __be16 h_vlan_proto;
    __be16 h_vlan_TCI;
};
```

Figure 4.2: VLAN–related members of `struct bpf_fib_lookup` that is used both for passing arguments to `bpf_fib_lookup` helper function as well as retrieving result of FIB lookup. Taken from [44, `include/uapi/linux/bpf.h`, Line 7211]

However, these fields are not populated during lookups – they are intentionally set to zero [44, `net/core/filter.c`, Line 5884]. The function correctly returns the VLAN interface index and MAC addresses, but since the VLAN interface does not implement XDP

transmit function, interface index of VLAN interface needs to be translated to underlying physical interface index. There are several possible approaches to implementing VLAN support:

- Create a BPF map that maps VLAN interface indexes to their VLAN IDs and physical interface indexes
- Modify the kernel to properly populate the VLAN fields in the `bpf_fib_lookup` structure
- Create new BPF helper functions specifically for retrieving VLAN IDs and physical interface indices – though this would be excessive since the structure for this information already exists in `bpf_fib_lookup`

4.2 Implementation

After discussing implementation strategies with the thesis supervisor and consultant, we implemented VLAN support in `xdp-forward` using two different approaches. The first approach uses a userspace-maintained mapping table, which works with unmodified kernels. The second approach relies on kernel modifications to properly populate the VLAN fields in the `bpf_fib_lookup` structure. Both implementations share common code for handling different VLAN translation scenarios.

Before implementing the actual VLAN handling logic, we extended the packet parsing code to identify VLAN-tagged frames. This is done by comparing the `EtherType` field to the `ETH_P_8021Q` macro. When a VLAN header is detected, the code sets a pointer `vhdr` to the VLAN header location and advances the offset to point to the inner protocol header, allowing the rest of the `xdp-forward` code to process it normally.

Both implementation approaches use the same code path for the three possible VLAN translation scenarios:

- For traffic between two VLAN interfaces (tagged to tagged), the code simply overwrites the existing VLAN header with the destination VLAN ID in following branch:

```
if (vhdr && fib_params.h_vlan_TCI)
    // packet has vlan header (vhdr) and has destination VLAN ID set
    ...
```

- When forwarding from a VLAN interface to an untagged interface, the code removes the VLAN header:

```
else if (vhdr && !fib_params.h_vlan_TCI)
    // packet has vlan header (vhdr), no VLAN ID for destination
    ...
```

- When forwarding from an untagged interface to a VLAN interface, the code adds a new VLAN header:

```
else if (!vhdr && fib_params.h_vlan_TCI)
    // packet has no~vlan header (vhdr), but has destination VLAN ID
    ...
```

The key difference between our two implementation approaches lies in how the VLAN information is supplied to these code paths. Both approaches need to populate the `h_vlan_TCI` and `h_vlan_proto` fields in the `struct bpf_fib_lookup` structure. By populating these fields before entering the VLAN handling code, we can use a unified approach for all scenarios, regardless of how the VLAN information was obtained.

Our extension of `xdp-forward` has been submitted as a merge request[1] to upstream project.

4.2.1 Userspace approach

The userspace approach to VLAN support relies on creating and maintaining a mapping between VLAN interface indices and their corresponding physical interfaces and VLAN IDs. This approach works with unmodified kernels.

Before loading the XDP program on physical interfaces, our implementation adds a step to discover VLAN devices configured on top of forwarding-enabled physical devices. This discovery is done by the `find_vlan_interfaces` function in the `xdp-userspace-vlans.c` file using the Netlink protocol[2].

For each physical interface, we retrieve information about all configured VLAN interfaces up to a predefined maximum of `MAX_VLANS_PER_IFACE`[3]. The function populates an array of custom-defined structures:

```
struct vlan_info {
   __u16 vlan_id; // VLAN ID
   int phys_ifindex; // Physical interface index
   int vlan_ifindex; // VLAN interface index
};
```

Figure 4.3: Structure for storing VLAN interface information. Structure used in map, which maps VLAN interface indices to their physical interfaces and VLAN IDs.

This structure contains all the information needed for VLAN translation: the VLAN ID itself, the physical interface index where the XDP program is loaded, and the VLAN interface index that the kernel uses for routing decisions. These structures are then stored in a BPF map of type `BPF_MAP_TYPE_HASH`, using the VLAN interface index as the key.

When the XDP program performs a routing lookup using `bpf_fib_lookup`, the kernel returns the destination MAC address and interface index where the packet should be forwarded. For VLAN interfaces, this returned index is the VLAN interface index, not the physical interface index needed for XDP redirection. To handle this, we implemented a translation function `set_vlan_params` which:

- Checks if the returned interface index exists in our VLAN map
- If found, updates the lookup parameters with the physical interface index and VLAN ID:

  ```
  fib_params->ifindex = vinfo->phys_ifindex;
  fib_params->h_vlan_TCI = vinfo->vlan_id;
  ```

[1] https://github.com/xdp-project/xdp-tools/pull/504

[2] A Linux kernel interface for communicating between kernel and userspace processes about network-related configuration and statistics.

[3] Currently, set to 16.

This translation is the key difference between the userspace approach and the kernel patch approach. The rest of the VLAN handling code (for adding, removing, or modifying VLAN tags) functions identically in both implementations.

A limitation of this userspace approach is that the VLAN mapping is created when `xdp-forward` is loaded and remains static afterward. If VLAN interfaces are added, removed, or modified, the XDP program must be unloaded and reloaded to update the mapping. A more robust solution would require a persistent userspace process that listens for Netlink messages about network configuration changes and updates the BPF map accordingly. Since this prolongs the packet path through `xdp-forward` BPF program, it adds processing overhead which results in to slightly lower performance (more on this, in separate Section 5.4.6). Therefore, we decided to activate this support only if `VLANS_USERSPACE` variable is set during compilation of `xdp-forward`.

4.2.2 Patched kernel approach

After confirming that our userspace solution worked with reasonable performance, this thesis' consultant Toke Høiland-Jørgensen decided to finish the kernel implementation for VLAN support in `bpf_fib_lookup`. We assisted with testing and debugging this kernel patch. This approach requires applying the patch to the kernel source code and recompiling the kernel, but provides a more integrated solution with better maintainability.

The key improvement in the kernel patch is addressing the two issues we identified with the current implementation:

- The kernel hardcodes 0 as the VLAN ID in the `bpf_fib_lookup` result structure
- It returns the VLAN interface index instead of the physical interface index

The consultant's patch modifies the kernel to correctly populate these fields, eliminating the need for the userspace-maintained mapping table. Our modification of `xdp-forward` adds a special flag passed to the `bpf_fib_lookup` helper function that activates this new behavior:

```
flags |= BPF_FIB_LOOKUP_RESOLVE_VLAN;
```

When this flag is set and the patched kernel is present, the lookup directly returns the physical interface index and the VLAN ID. The consultant plans to submit his patch upstream to the Linux kernel.

Since this approach needs to do the same packet manipulations as userspace approach, it slightly extends the code path in the XDP program and adds a small overhead. We implemented a compile-time flag `VLANS_PATCHED` that enables the kernel patch integration. This allows users to select the appropriate implementation based on their kernel version. It's important to note that even with the kernel patch, the packet manipulation logic (adding, removing, or modifying VLAN tags) remains identical to the userspace approach. The patch only replaces the need for the userspace map, simplifying the implementation and improving maintainability.

4.3 Functional tests

In this section, we will evaluate functionality of our solutions. This includes all possible directions and type of traffic, as we mentioned earlier and both versions of our implementation – userspace and kernel patched.

For both implementations we ran tests more described in Section 5.4.6 with our benchmarking tool with `--functional` argument, which sets up whole test bed and stops before running the performance test waiting for user input before continuing. Instead, we have started `tcpdump`[4] on receiver's NIC with:

```
$ tcpdump -i ens2f1np1 -B 4096 -w log.pcap
```

Then, we generated a couple[5] of packets towards receiver's NIC with `xdp-forward` on forwarder.

Non trunked to non trunked In this test case, we are forwarding regular traffic without any VLAN tag to be sure, that our solutions do not break forwarding of this type of traffic.

```
$ tcpdump -nn -e -r untagged2untagged.pcap
14:16:05.152550 src > dst, ethertype IPv6 (0x86dd), length 64:
src_ip.12000 > dst_ip.12000: UDP, length 2
```

As we can see, the packet arrived as well as there is no VLAN header. Thus, our implementation does not break regular traffic forwarding.

Non trunked to trunked This test case is supposed to verify, that `xdp-forward` can add VLAN header when packet is forwarded towards VLAN network.

```
$ tcpdump -nn -e -r untagged2tagged.pcap
14:19:35.929659 src > dst, ethertype 802.1Q (0x8100), length 68: vlan 20,
p 0, ethertype IPv6 (0x86dd), src_ip.12000 > dst_ip.12000: UDP, length 2
```

As we can see, received packet has the correct VLAN header, as well as VLAN ID. Therefore, our solutions are able to insert VLAN header.

Trunked to non trunked In this test case, we will test if our solutions are capable of removing VLAN header.

```
$  tcpdump -nn -e -r tagged2untagged.pcap
14:17:46.086632 src > dst, ethertype IPv6 (0x86dd), length 64:
src_ip.12000 > dst_ip.12000: UDP, length 2
```

We have sent VLAN tagged packet and packet with no VLAN header arrived. Therefore, our solutions are capable of removing VLAN headers.

Trunked to trunked In this test case, we well verify, that our solutions correctly rewrites VLAN ID when forwarding packets between two networks using VLANs.

```
$  tcpdump -nn -e -r tagged2tagged.pcap
14:21:19.879715 src > dst, ethertype 802.1Q (0x8100), length 68: vlan 20,
p 0, ethertype IPv6 (0x86dd), src_ip.12000 > dst_ip.12000: UDP, length 2
```

As we can see, received packet has correct VLAN ID set.

[4] https://www.tcpdump.org/manpages/tcpdump.1.html
[5] The amount of packets is not important, all the packets are processes the same way.

Chapter 5

Performance analysis

This section is dedicated to comparing the performance of packet forwarding between the standard Linux kernel stack and XDP-based forwarding (`xdp-forward`) that we described earlier in Section 2.6.4. To properly benchmark these two approaches, we needed a good testing methodology and appropriate tools.

5.1 Networking performance measurement tools

We first looked at the XDP test suite[1] which was a bachelor thesis focused on creating testing framework for XDP. However, this suite mainly focuses on functional testing rather than performance testing. It uses hardcoded BPF programs for its tests [22], and adapting it for performance measurements would require significant amount of changes to this project.

Instead, we decided to use the Linux Network Stack Test (LNST) framework[2], which is designed specifically for testing network performance. We extended this framework to test both regular kernel forwarding and `xdp-forward`, allowing us to directly compare their performance.

In the following sections, we'll describe the tools we used for packet generation (pktgen), xdp-bench, and the LNST framework as well as the actual test case.

5.1.1 pktgen

Pktgen is a high-performance packet generator integrated directly into the Linux kernel. The in-kernel implementation provides distinct advantages over userspace alternatives, particularly in terms of performance and compatibility. Being part of kernel allows it to bypass most of the Linux networking stack, offering direct access to the host system's network interface driver and transmission process. [33]

It can generate traffic belonging to a single flow by maintaining consistent source and destination parameters, or it can simultaneously create multiple flows by manipulating the 5-tuple connection parameters (source IP, destination IP, source port, destination port, and protocol).

Implementation of pktgen requires minimal setup, as it exists as a kernel module within the standard Linux kernel. Because of mentioned advantages LNST has chosen to use `pktgen` for test cases that require a high performance packet generator. However, current implementation of wrapper around `pktgen` in LNST does not support multiple flows,

[1] https://github.com/shoracek/xdp-test-suite/ – repository last updated 5 years ago
[2] https://lnst.readthedocs.io/en/latest/

or setting rate of generated packets. Therefore, we had to extend LNST's implementation of wrapper to support multiple flows. These changes are required for our following test case implementation as well, they might be useful for upstream community as well. Thus, we opened a merge request with our changes to upstream LNST project[3].

5.1.2 Linux Network Stack Test – LNST framework

The Linux Network Stack Test (LNST) framework is a test automation and orchestration tool specifically designed for network testing across multiple hosts. LNST establishes direct communication channels with devices that are included in test scenario. The framework handles the entire test lifecycle, including initial setup, test execution, result synchronization, and system restoration after testing completion.

A significant advantage of LNST is its integration with established networking performance tools. Rather than reimplementing measurement functionality within Python, LNST leverages mature tools such as `iperf3`, `pktgen`, and `xdp-bench` as execution backends. This approach abstracts the underlying tools while maintaining their native performance characteristics, allowing the framework to gather and consolidate results for comparative analysis.

Recipes

Recipes in LNST define the complete test scenario, including network configuration, test execution, and evaluation of results. Each recipe runs measurements using the tools mentioned above, configures evaluators, and processes test results.

LNST recipes typically implement the following methods:

- Test wide configuration – Sets up the complete test environment including network interfaces, routing tables, and any system parameters required for testing
- Ping tests – Run on agent machines to verify basic connectivity between systems under test before proceeding with performance measurements
- Performance tests – Execute the actual measurements, which involves configuration of test parameters, running the selected tool, gathering results, and evaluating performance metrics
- Test wide deconfiguration – Restores all systems to their original state by removing configuration changes made during testing

5.2 Testbed

The testbed comprises various CPU and NIC configurations commonly used in high performance networking environments. NICs were selected with different speeds: Intel E810 series (25 Gbps), Mellanox ConnectX and Solarflare cards (100 Gbps), and Broadcom BCM57508 (200 Gbps). Testing across multiple CPU architectures (Intel, AMD, ARM) and network drivers provides insight into architecture-specific performance characteristics. While direct comparisons between different hardware platforms have limitations, these configurations help establish performance baselines across varying environments.

[3] https://github.com/LNST-project/lnst/pull/400

Processor	Sockets	Cores	NIC	Alias
Intel Xeon 6438N @ 2.0GHz	2	32	MT2910 [CX-7]	Intel1, mlx5
			Intel E810-XXV	Intel1, ice
AMD 7443P @ 2.8GHz	1	24	BCM57508	AMD, bnxt_en
ARM Altra Q80-30 @ 3.0GHz	1	80	MT2894 [CX-6]	ARM, mlx5
			Intel E810-C	ARM, ice
Intel Xeon 5520+ @ 2.2GHz	2	56	Solarflare SFC9250	Intel2, sfc

Table 5.1: Testbed CPUs and NICs used for comparing driver performance. Alias column represent names of individual configuration in results section.

5.3 Test scenario design

This section describes the implementation of performance measurement tests using the LNST framework. We developed two test recipes: `ForwardingRecipe` for measuring standard Linux kernel forwarding performance and `XDPForwardingRecipe` for measuring `xdp-forward` forwarding performance. With these changes, we have opened another merge request with additional changes to upstream LNST project[4].

The test scenario requires two physical hosts, each equipped with two network interfaces. As illustrated in Figure 5.1, traffic flows from the first host (generator) to the second host (forwarder) and then back to the first host (receiver). To prevent local delivery shortcuts on the first host, we isolate each network interface in separate network namespaces.

Our test topology uses IPv6 and IPv4 subnets which are further divided into smaller subnets (depending on, what IP version is test run for). Two transit networks are established, one connecting the generator to the forwarder, and another connecting the forwarder to the receiver. This design mimics real-world network configurations in which traffic flows from core routers (represented by the generator and forwarder) to an edge router (represented by the receiver) serving client networks. In our test scenario, packets that reach the receiver are counted and then dropped rather than delivered to actual clients.

5.3.1 Test environment configuration

Before executing any performance tests, we must establish a properly configured test environment. This section examines the implementation of the `test_wide_configuration` function in our LNST recipe, which handles all necessary networking configuration.

The configuration process implements several steps to prepare the test environment:

- Enabling IP forwarding on the forwarder host by setting the appropriate sysctl values.
- Subnetting the primary test networks (passed to recipe by `net_ipv4` or `net_ipv6` parameters) into three smaller networks:
  - An egress network connecting the generator to the forwarder
  - An ingress network connecting the forwarder to the receiver
  - A routed network space serving as destination networks
- Creating and configuring network namespaces on the first host to isolate traffic generation from reception

[4] https://github.com/LNST-project/lnst/pull/402

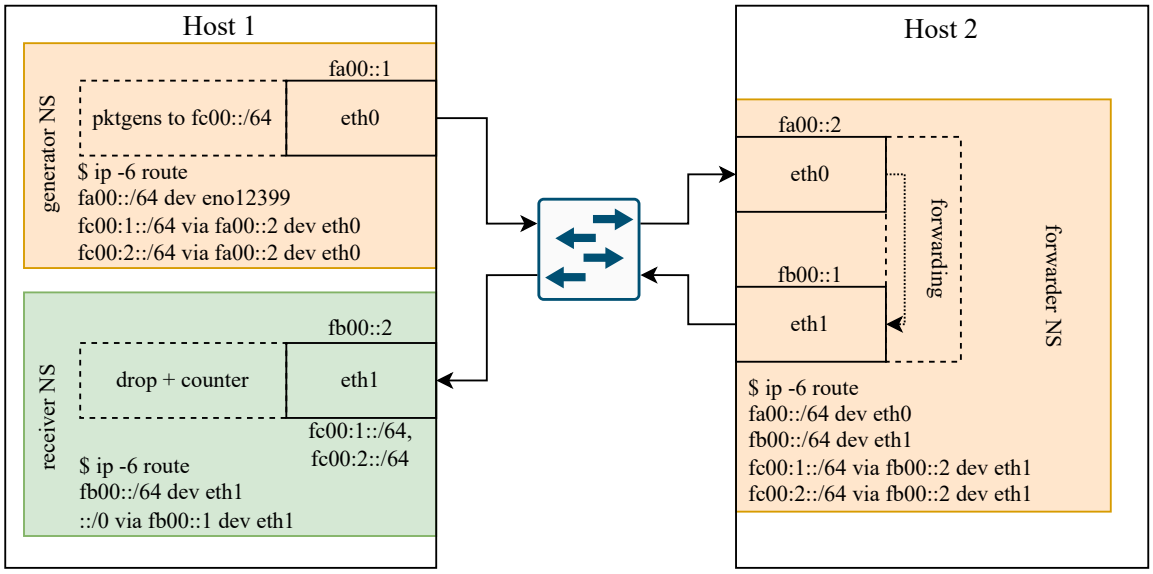

Figure 5.1: Figure illustrates test scenario used for gathering results. `pktgen` generates packets within generator's namespace. These are sent over the switch towards forwarder, which does the routing table lookup and forwards packets to receiver namespace using different NIC, based on routing table.

- Configuring transit IP addresses on all interfaces to establish connectivity between test components
- Setting up static routes to ensure packets follow the intended forwarding path since there is no dynamic routing protocol in place

When testing is complete, the `test_wide_deconfiguration` function reverses all these changes, returning the system to its original state by removing static routes and disabling IP forwarding.

The `XDPForwardingRecipe` extends this configuration by enabling `xdp-forward`. Similarly, during deconfiguration, the XDP program is unloaded from these interfaces to restore the original state.

5.3.2 Verifying environment configuration

After establishing the test environment, the recipe executes a series of ICMP ping tests to ensure proper routing configuration between all test components. The `generate_ping_endpoints` method defines three ping paths:

- From generator to forwarder
- From forwarder to receiver
- From generator to receiver (testing end-to-end forwarding path)

Since each ICMP echo request generates a corresponding reply packet, these unidirectional ping tests inherently verify bidirectional connectivity.

5.3.3 Performance measurements

After establishing and verifying the test environment, the recipe executes a series of performance measurements to evaluate forwarding performance.

The following sections detail the specific implementation and methodology for each measurement category.

Forwarding performance measurement

The core forwarding performance measurement is implemented in the `ForwardingMeasurement` class.

At the generator side, the measurement uses the LNST's `pktgen` module to generate packets. These packets are crafted with destination IP addresses from the configured destination networks, but with the MAC address set to the forwarder's ingress interface, where are forwarder to receiver. At the receiver side, we deploy `xdp-bench` in drop mode, it simply counts incoming packets and then drops them immediately. This avoids packet being processed by kernel, which is demands more resources.

The measurement also employs `InterfaceStatsMonitor` on the forwarder's interfaces to sample NIC hardware counters[5]. While these interface statistics serve as supplementary data points. Our experiments indicate that some NICs drivers (especially `mlx5_core` driver) do not update standard counters when running XDP programs, instead updating non-standard counters only accessible via `ethtool -S` only.

Monitoring CPU utilization

To measure the CPU resources consumed during forwarding operations, our test employs the `CPUStatMonitor` class from the LNST framework[6]. This monitoring component samples CPU statistics at regular intervals by reading data directly from the `/proc/stat` file. The monitor collects comprehensive CPU metrics including user time, system time, idle time, interrupt handling time, and I/O wait time. Each sample is timestamped and the difference between consecutive samples provides precise CPU utilization measurements during test execution. In our tests, samples are collected at one-second intervals, providing sufficient granularity.

5.3.4 Multiple CPUs fowarding performance

This test scenario evaluates how forwarding performance scales across multiple CPU cores, which helps identify potential bottlenecks in both kernel-based and XDP-based forwarding. To ensure fair distribution of processing load, we configured the test environment to direct specific network flows to designated CPU cores using the `DevFlowsPinningHWConfigMixin` class. This approach leverages hardware flow steering capabilities to map each flow to a specific RX queue and its associated CPU core, rather than relying on standard Receive Side Scaling (RSS).

For the test, we incrementally increased the number of active CPU cores from one to the count of CPUs of our system, measuring forwarding performance at each step. All other system parameters remained constant.

[5] Exported by `sysfs` in `/sys/class/net/<NIC>/statistics/`.

[6] https://github.com/LNST-project/lnst/blob/master/lnst/Tests/CPUStatMonitor.py

5.3.5 RX/TX ring size performance comparison

Ring buffer sizes can impact forwarding performance, particularly for batch processing behaviors. This test investigates how varying these buffer sizes affects XDP-based forwarding performance. Since XDP has no other buffers nor queues, its performance is might be particularly sensitive to ring buffer configuration. Therefore, we focused this test specifically on the XDP forwarding implementation.

We configured xdp-forward with various RX and TX ring size combinations, ranging from minimum supported values to the maximum allowed by the hardware. Performance was measured in terms of maximum packet rate.

5.3.6 CPU utilization scaling for various packet rates

Another test case examines how CPU utilization scales with increasing packet rates for both forwarding implementations. Instead of testing all possible rates, we measured at intervals of 30,000 packets per second from 30,000 to 480,000 pps. On top of that, we added the zero-drop rates identified in Section 5.3.8. For each rate, we recorded CPU statistics. These measurements reveal efficiency differences between kernel-based and XDP-based forwarding under varying network loads.

5.3.7 Forwarding performance for various packet sizes

Packet size significantly impacts forwarding performance and CPU utilization. While 64-byte packets stress forwarding planes, real-world traffic includes diverse packet sizes. AMS-IX statistics[7] show this distribution:

- 64 – 127B: 27.8%
- 128 – 255B: 5.5%
- 256 – 511B: 2.8%
- 512 – 1023B: 3.5%
- 1024 – 1513B: 37%
- Larger packets: 23%

Based on this and RFC 2544 [10], we selected packet sizes of 64, 128, 256, 512, 1024, 1280 and 1518 bytes for our analysis.

For each size, we measured CPU utilization at full throughput capacity, revealing the actual forwarding performance for both kernel and XDP forwarding implementations across different packet sizes.

5.3.8 Drop rate measurements

To determine the maximum sustainable packet forwarding rate, we implemented a binary search algorithm in the `NoDropRateMixin` class. This approach systematically identifies the highest packet rate that can be processed without exceeding a specified packet drop threshold.

[7] https://stats.ams-ix.net/sflow/size.html?type=size

The algorithm works by generating traffic with `pktgen` while monitoring packet counts at interfaces along the path. It employs an iterative process:

- If the measured drop rate exceeds the configured threshold, the packet generation rate is reduced
- If the drop rate remains below the threshold, the generation rate is increased
- Each time the direction changes (from increasing to decreasing or vice versa), the step size is halved
- Each iteration involves a 5-second measurement period, with an additional period skipped after rate changes to allow stabilization

This implementation, while not as sophisticated as dedicated tools like `TRex`[8], provides valuable insight into forward performance capacity. We selected this approach for its compatibility with test automation environments, where installing specialized software packages can be problematic[9]. The method is particularly useful for comparative testing between kernel forwarding and XDP forwarding implementations, even though absolute measurements may be affected by packet reordering and timing variations.

5.3.9 VLAN test scenarios

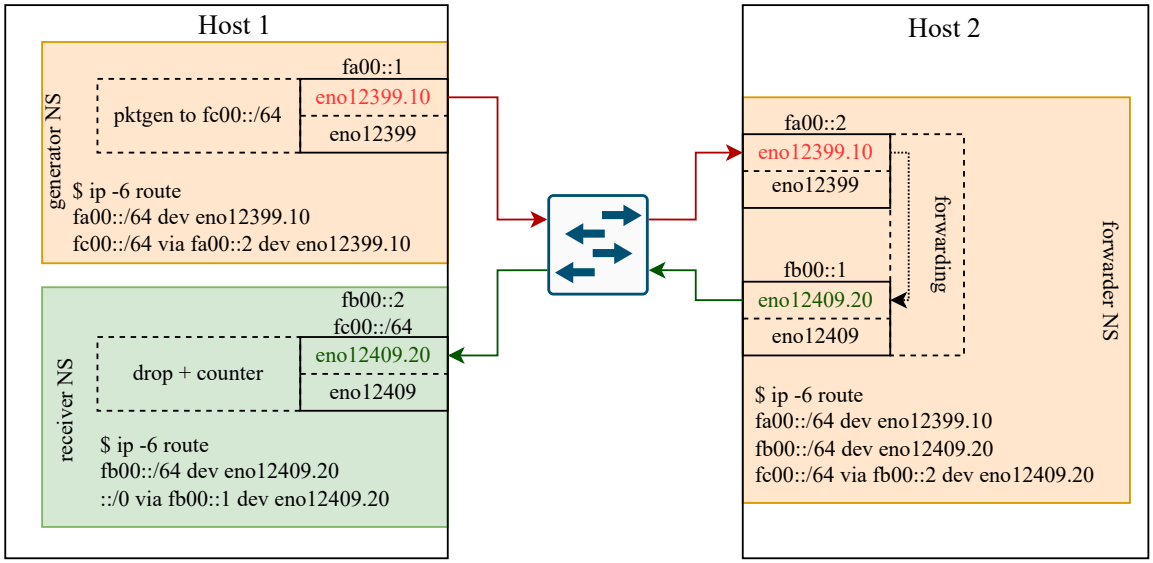

Figure 5.2: Figure shows the basic VLAN test, where packets are generated within VLAN tagged network (VLAN ID 10) and sent towards VLAN enabled network at receiver side (VLAN ID 20) via forwarder.

To evaluate VLAN processing performance, we extended our forwarding test methodology to incorporate various VLAN configurations. The `VlansForwardingMixin` class extends

[8] https://trex-tgn.cisco.com/

[9] For example, `mlx5_core` driver requires additional package to work with `TRex` [45]. Another problem, we faced is, that the latest version of `TRex` does not work with newer Python versions than 3.11 [27].

the basic `ForwardingRecipe` and `XDPForwardingRecipe` by overriding the test configuration method to create VLAN devices on top of the physical interfaces.

Our VLAN testing methodology implements three distinct test scenarios, each evaluating a different aspect of VLAN processing:

- Tagged to tagged — As illustrated in Figure 5.2, traffic originates from a VLAN-tagged network and is directed toward another VLAN-tagged network. This scenario tests the performance impact of swapping VLAN tags during forwarding, which requires the forwarder to process both ingress and egress VLAN headers.
- Tagged to untagged — Traffic originates from a VLAN-tagged network but is destined for a network that does not use VLANs. This scenario evaluates the performance of VLAN tag removal during forwarding, which occurs when traffic moves from a segregated VLAN environment to a flat network.
- Untagged to tagged — Traffic originates from a network without VLAN tagging but is destined for a VLAN-enabled network. This scenario tests the performance of adding VLAN tags during forwarding, which happens when traffic from a flat network enters a VLAN-segmented environment.

5.4 Results

In this section, we will focus on results evaluation of tests, we described earlier. The results are based on 10 subsequent iterations of test case running on one of the machine pair mentioned in Section 5.2. Unless presenting results for CPU utilization focused test cases, all measurements were bottlenecked by the CPU with utilization approximately 100 % per core. This section mostly includes plots as they are easier to read. However, full-sized results in table can be found in Appendix B.

5.4.1 Driver performance comparison

As we mentioned earlier, XDP is implemented within NIC driver itself. Therefore, it makes sense to start with comparison of individual driver performance.

RX/TX ring size

Before digging into advanced results, we focused on experimenting with `RX` and `TX` ring sizes. Table 5.2 shows comparison of forwarding performance when using the XDP-based forwarding plane for multiple drivers. In this test case, we started at ring sizes of 1024 and we were increasing it by step size of 1024 with end at driver's maximum value.

RX/TX ring size	bnxt_en	sfc	ice	mlx5
1024	$2.0\,M \pm 10.0\,K$	$1.2\,M \pm 5.2\,K$	$4.1\,M \pm 9.6\,K$	$2.6\,M \pm 2.9\,K$
2047	$2.0\,M \pm 48.7\,K$			
2048		$1.2\,M \pm 5.8\,K$	$4.1\,M \pm 9.7\,K$	$2.6\,M \pm 3.5\,K$
3072			$4.1\,M \pm 31.3\,K$	$2.6\,M \pm 2.7\,K$
4096			$3.9\,M \pm 8.7\,K$	$2.6\,M \pm 3.0\,K$
5120			$4.1\,M \pm 8.1\,K$	$2.6\,M \pm 5.4\,K$
6144			$4.1\,M \pm 6.2\,K$	$2.6\,M \pm 3.1\,K$
7168			$4.0\,M \pm 8.1\,K$	$2.7\,M \pm 4.5\,K$
8160			$4.0\,M \pm 30.6\,K$	
8192				$2.7\,M \pm 5.0\,K$

Table 5.2: The table presents comparison of various RX and TX ring size configurations across various drivers. The measurements were done using `IPv6` and `xdp-forward` since we expect XDP to be most affected by these settings. *Empty values represents values incompatible with driver. E.g. ice's maximum ring sizes are 8160.*

From Table 5.2 we can see, that various ring size configurations has little to no effect on forwarding performance. Therefore, for following tests we stick to the driver's default values. It might has *some* impact in cases, where packet rate is not constant as it was in our use-case. However, our test is a provides first insights into impact of ring size.

Single stream

This section presents results of forwarding performance on both forwarding planes we focused on – kernel and `xdp-forward`. In this test case, we focused on single stream forwarding performance and we compare individual drivers results to each other in Figure 5.3.

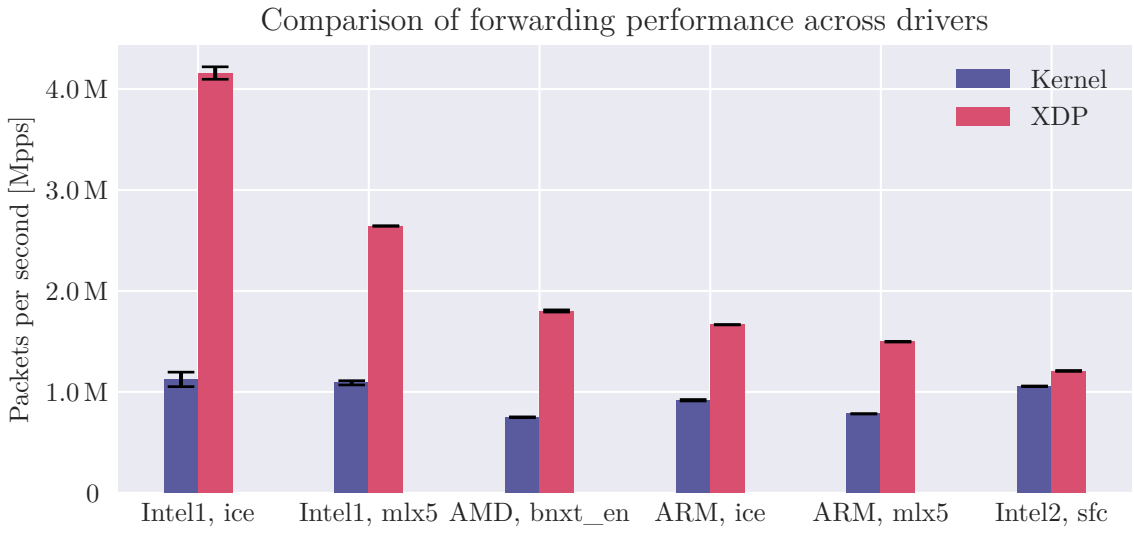

Figure 5.3: Comparison of forwarding performance across different network drivers and hardware platforms. The figure shows packet forwarding rates measured in packets per second (Mpps) for both standard kernel forwarding (blue) and `xdp-forward` (red) forwarding planes. Error bars indicate standard deviation across test iterations. The results demonstrate that `xdp-forward` consistently achieves higher forwarding rates than kernel forwarding across all tested configurations.

Figure 5.3 and Table B.1 demonstrate that `xdp-forward` outperforms kernel forwarding across all tested configurations, though with varying improvement ratios. The Intel1 platform with the ice driver exhibits the highest performance differential, while the Intel2 platform with the sfc driver shows the smallest gap (1.4 *Mpps* vs 1.1 *Mpps*, about 1.3× improvement).

IPv6 traffic generally shows lower performance than IPv4, with reductions ranging from approximately 2% to 28% depending on the configuration. This performance difference is more pronounced in `xdp-forward` implementations.

It should be noted that direct comparisons should only be made between different drivers on the same hardware platform due to varying processor capabilities across test systems. Despite this limitation, these results consistently demonstrate the performance advantage of `xdp-forward` across diverse hardware environments.

In following section, we will focus on presenting results of Intel1, mlx5 configuration only, as we identified this one as widely used.

Multiple streams

After analyzing single stream performance, we extended our evaluation to assess how both forwarding planes scale with multiple CPU cores and parallel traffic streams. For these tests, the kernel forwarding configuration used the `noqueue` queuing discipline and had `netfilter` completely disabled because we encountered scaling limitations when attempting to exceed 10 Mpps (further details in Appendix A). This configuration makes the comparison with `xdp-forward` more comparable, as XDP naturally operates without queuing mechanisms or `netfilter` functionality.

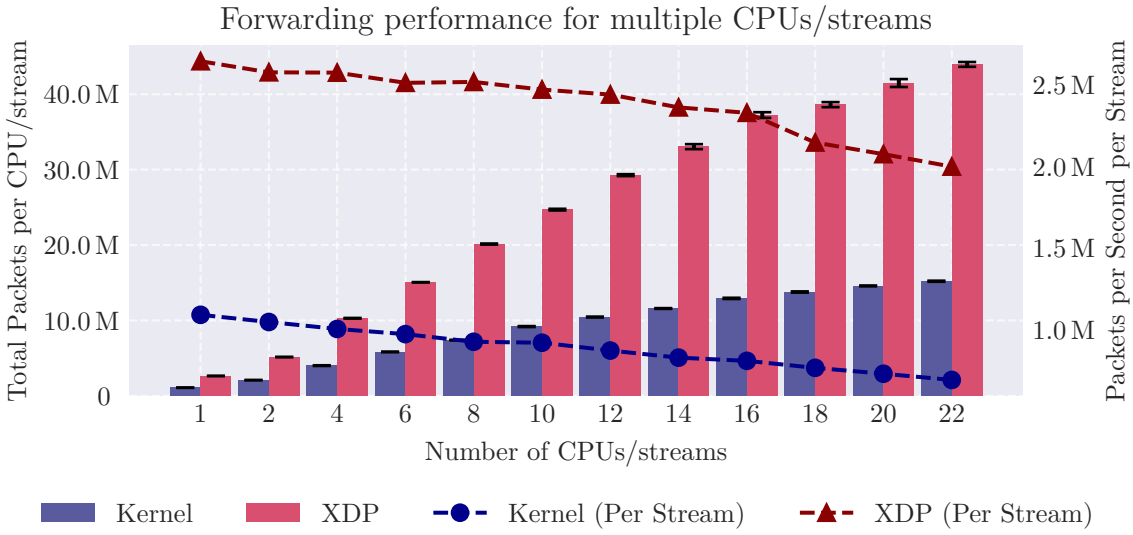

Figure 5.4: Forwarding performance scaling with increasing number of CPU cores and traffic streams. The left y-axis and bar charts show total packets forwarded per second across all streams, while the right y-axis and line plots represent per-stream forwarding rates. `xdp-forward` consistently delivers higher throughput than kernel forwarding across all configurations. Both forwarding implementations demonstrate scaling with additional CPUs.

Figure 5.4 and Table B.2 demonstrate that both forwarding implementations scale effectively with additional CPU cores and streams. The total forwarding throughput increases nearly linearly as more processing resources are allocated. At the maximum tested configuration of 22 streams, kernel forwarding achieves approximately 15.2 Mpps for IPv4, while `xdp-forward` reaches 47.5 Mpps – maintaining its significant performance advantage with a 3.1× improvement ratio.

Regarding IPv6 traffic, we observe that it scales at a comparable or slightly higher pace than IPv4 for both forwarding planes. This result might be most likely explained by the lower absolute packet rates for IPv6.

5.4.2 Drop rate stability

For this test we used the methodology outlined in Section 5.3.8, employing a binary search algorithm to find the maximum packet rate that achieves a specified target drop percentage. Figure 5.5 and Table B.3 present the results across various desired drop rates.

The kernel forwarding implementation demonstrates notably consistent behavior with standard deviations. This stability persists even with the `noqueue` queuing discipline enabled and `netfilter` disabled. In contrast, `xdp-forward` exhibits substantially higher variability.

This disparity in stability most likely comes from fundamental architectural differences. In contrast, `xdp-forward` operates with minimal buffering. When a packet arrives, it must be processed immediately within the driver context. If resources are temporarily unavailable—for instance, if the transmit device is busy—the current packet must be dropped. This immediate processing model improves the latency for successfully forwarded packets but creates significant variability in drop behavior under load.

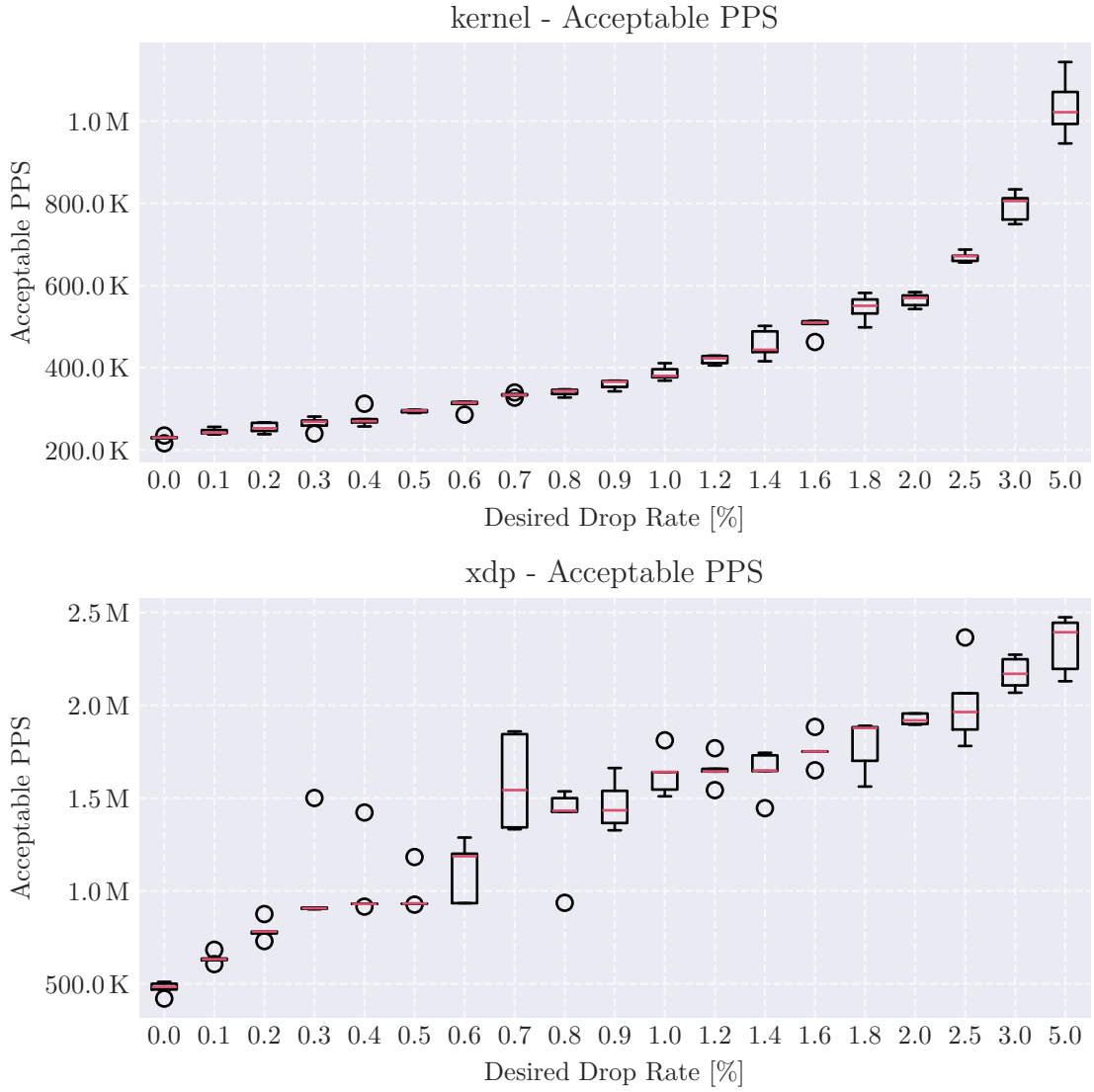

Figure 5.5: Comparison of acceptable packet rates for varying desired drop rates between kernel and `xdp-forward` forwarding planes. The box plots represent the distribution of acceptable PPS values across multiple test iterations for each target drop rate. The kernel forwarding results (top) show remarkably consistent performance with minimal variance between iterations, while `xdp-forward` results (bottom) exhibit significantly higher variability. This difference highlights the stability characteristics of buffered versus immediate packet processing approaches.

The absolute performance difference remains consistent with our previous findings, with `xdp-forward` handling approximately $2-2.3\times$ more packets than kernel forwarding before reaching comparable drop rates. These results further reinforce the potential benefits of implementing queuing mechanisms within XDP, as discussed in Section 3.4.

Due to the computationally intensive nature of these tests, we limited this experiment to five iterations per configuration, which provides useful insights into the relative stability characteristics of both forwarding mechanisms anyway.

5.4.3 CPU Utilization for Packet Rate Scaling

While our previous section focused on presenting results of the maximum packet rates achievable with specific drop percentages, this section examines how CPU utilization scales with increasing packet rates. Rather than testing all possible rates up to the maximum, we selected discrete test points ranging from 30 000 packets per second up to 484 522 PPS (which represents the average 0 % drop rate from the previous section with outliers removed), using increments of 30 000 PPS and including 230 190 (which represents average 0 % drop rate for kernel).

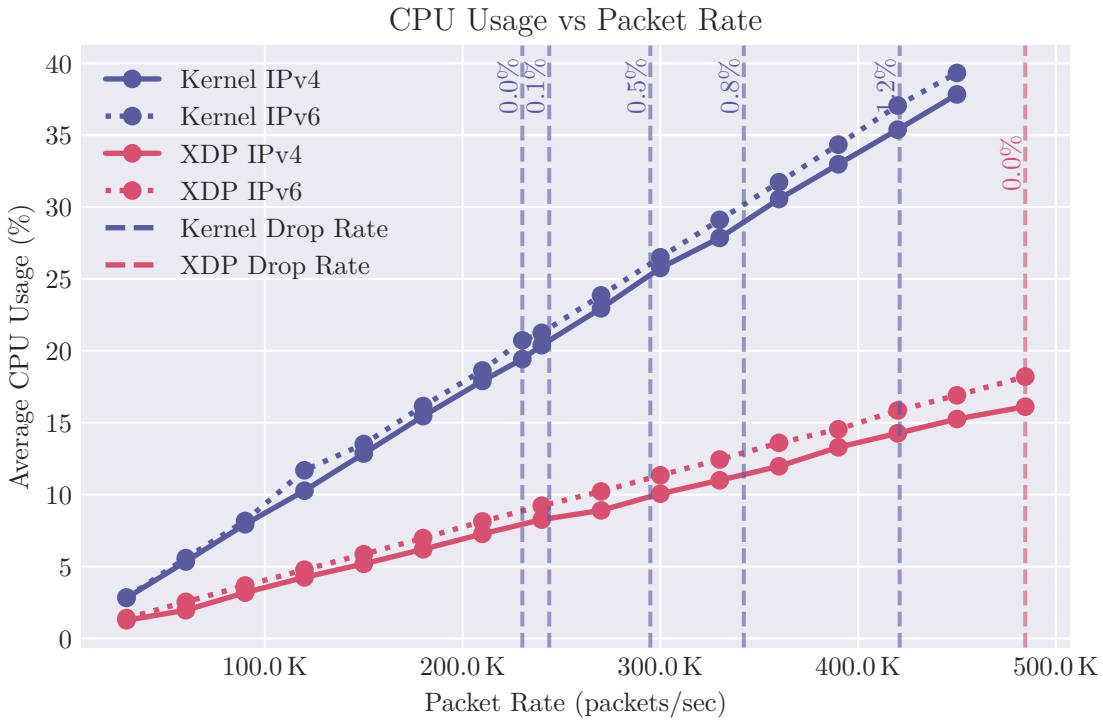

Figure 5.6: CPU utilization scaling with increasing packet rates for kernel and `xdp-forward` implementations. The graph plots average CPU usage (y-axis) against packet rate (x-axis) for both IPv4 and IPv6 traffic. Vertical dashed lines indicate packet rates where measured drop percentages occur. The significantly lower slope of the `xdp-forward` curves demonstrates its superior CPU efficiency compared to kernel forwarding.

As shown in Figure 5.6 and detailed in Table B.4, CPU utilization scales linearly with packet rate for both forwarding planes. However, the slope of this relationship differs signif-

icantly between implementations. The kernel forwarding implementation exhibits a steeper gradient.

This efficiency difference is further quantified in the „Ratio“ columns of Table B.4, which represent packets processed per percentage of CPU utilization. The kernel maintains a relatively consistent ratio of approximately 11,800 packets per CPU percentage point for IPv4, while `xdp-forward` achieves ratios typically exceeding 29,000 packets per CPU percentage point—a 2.5× efficiency advantage.

For IPv6 traffic, both forwarding planes show slightly higher CPU utilization compared to their IPv4 counterparts, reflecting the additional processing overhead associated with larger IPv6 headers.

5.4.4 Zero Drop CPU Utilization Scaling

This section presents results of CPU utilization when scaling the number of processing cores while maintaining zero packet loss (using the packet rate, we measured earlier). As shown in Figure 5.7 and Table B.5, both forwarding implementations achieve linear scaling in terms of total packets processed, with `xdp-forward` consistently handling approximately twice the packet volume of kernel forwarding.

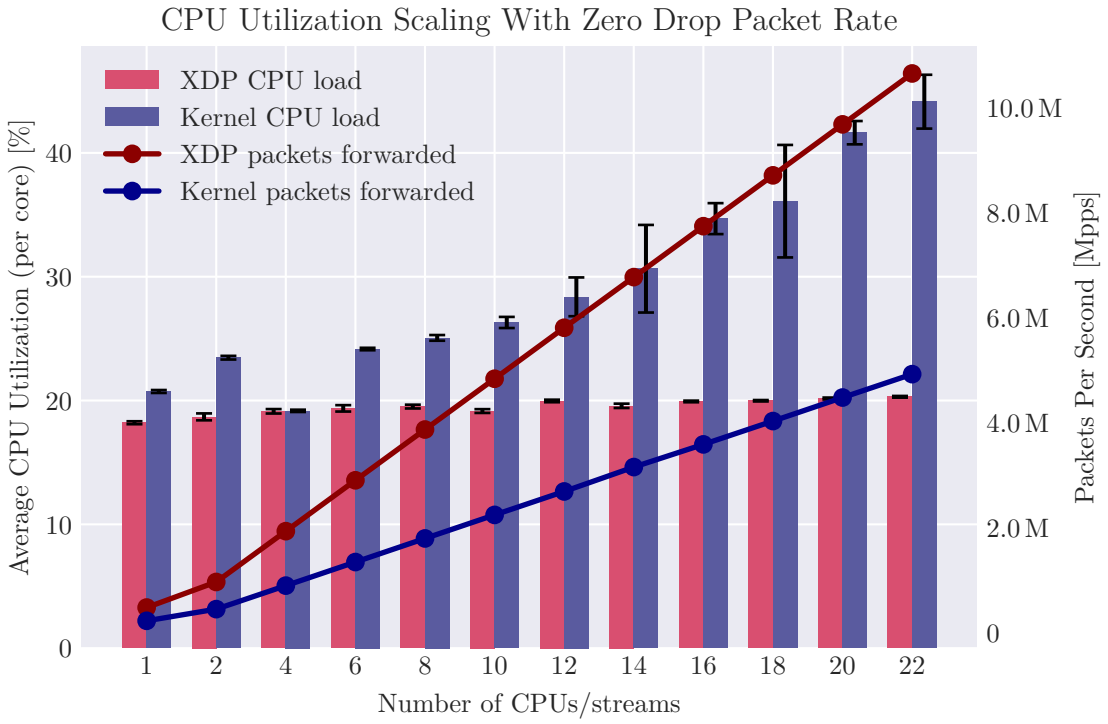

Figure 5.7: IPv4 zero-drop rate performance scaling with increasing core count. Bar charts show per-core CPU utilization, while lines represent total throughput. `xdp-forward` maintains consistent CPU usage per core while kernel forwarding shows increasing utilization overhead as cores are added, despite both achieving linear packet throughput scaling.

The key difference appears in per-core CPU utilization patterns. `xdp-forward` maintains nearly constant per-core CPU usage (16-20%) regardless of the number of cores, indicating excellent scaling efficiency. In contrast, kernel forwarding shows steadily increas-

ing per-core utilization as cores are added, growing from approximately 20% with a single core to over 40% with 22 cores for IPv6 traffic.

The „Overhead“ column in Table B.5 quantifies this effect, showing that `xdp-forward`'s additional per-core CPU overhead remains below 2.5% even at 22 cores, while kernel forwarding reaches over 20% added overhead. This behavior aligns with our findings in Appendix A.

5.4.5 Packet size scaling

This section extends our evaluation to examine how forwarding performance varies with packet sizes ranging from 64 to 1518 bytes. As shown in Figure 5.8 and Table B.6, the two forwarding planes behaves differently to packet size variations.

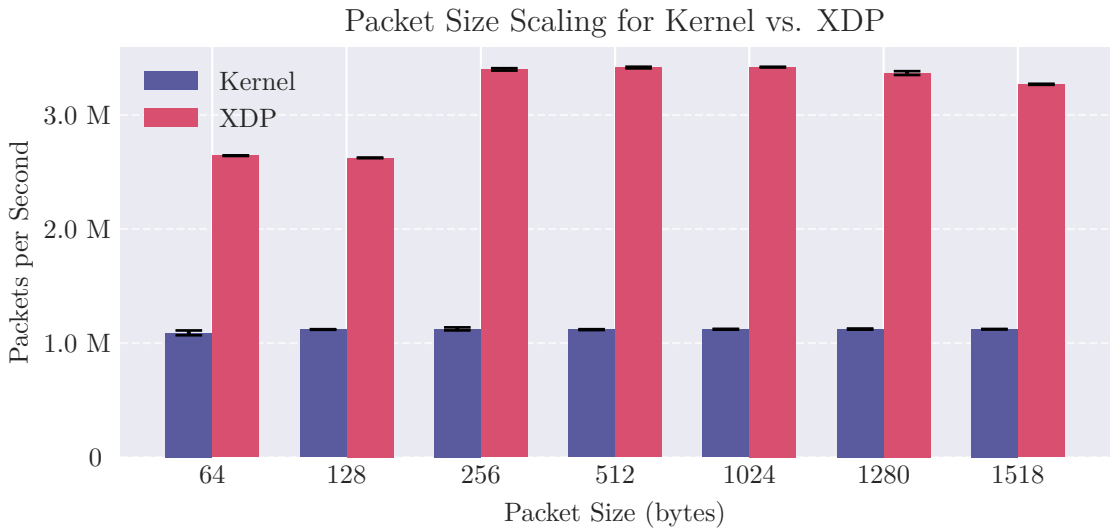

Figure 5.8: Packet forwarding performance comparison across different packet sizes using the Intel,mlx5 driver with IPv6 traffic. The graph shows packet processing rates for both kernel forwarding (blue) and `xdp-forward` (red). While kernel maintains consistent packet rates regardless of size, `xdp-forward` shows size-dependent performance variation, with optimal throughput at medium packet sizes.

Kernel forwarding demonstrates consistent processing rates across all sizes tested. For the Intel,ice platform with IPv4 traffic, it consistently processes approximately 1.3 $Mpps$ regardless of size. Similarly, IPv6 traffic rates remain stable at around $1.1 - 1.2\,Mpps$.

However, `xdp-forward` shows performance variations with packet size. On the Intel,mlx5 platform, it achieves optimal performance with medium-sized packets (256-1024 bytes), reaching $3.7 - 3.8\,Mpps$ for IPv4.

5.4.6 VLANs

This section is dedicated to present results of our VLAN tests. Firstly, we will look into comparing forwarding performance with VLAN offloads enabled and disabled, then we will present results of various VLAN test cases to test out our `xdp-forward` extension.

Offloaded VLAN tagging

Before evaluating our VLAN extension for `xdp-forward`, we first assessed the impact of VLAN processing on kernel forwarding performance. Modern NICs offer VLAN offloading capabilities to handle tag insertion and removal in hardware, but these offloads cannot be used with XDP.

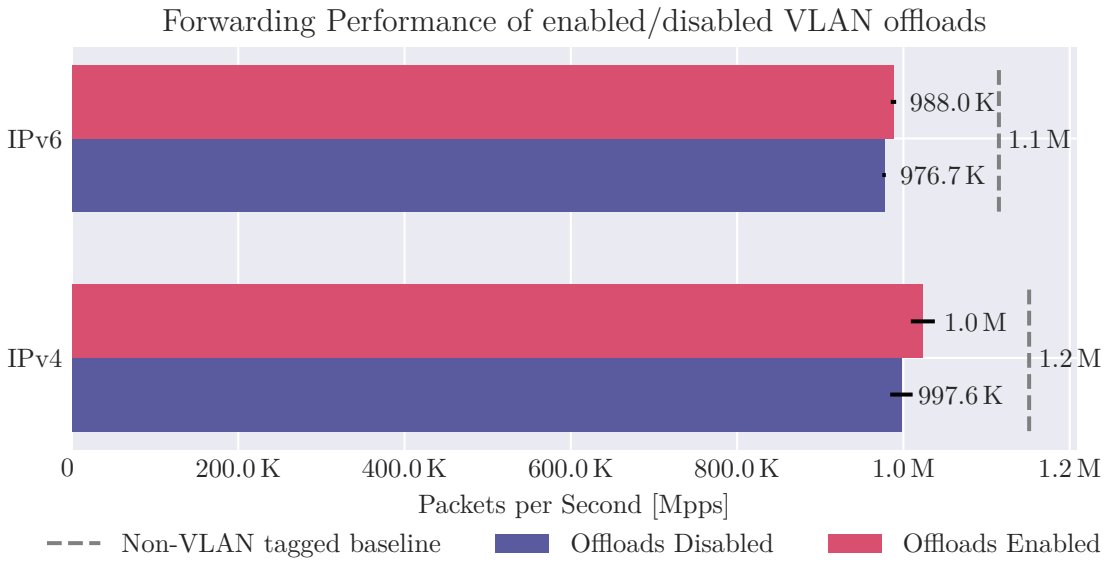

Figure 5.9: Performance comparison of kernel forwarding with different VLAN handling approaches. The graph shows packet forwarding rates for baseline (no VLAN tags), hardware-offloaded VLAN processing, and software-based VLAN processing for both IPv4 and IPv6 traffic. VLAN processing reduces throughput by approximately 11%.

As shown in Figure 5.9 and Table B.7, VLAN-tagged packets reduce kernel forwarding performance by 11-13% compared to non-tagged packets. However, the performance difference between hardware-offloaded VLAN handling and software-based processing is only about 1-2%. This suggests that while VLAN processing does affect performance, the actual mechanism of tag handling contributes little to the overall overhead.

Various Traffic Type Comparison

This section evaluates performance across four different VLAN traffic patterns: untagged-to-untagged (no VLANs), tagged-to-tagged, untagged-to-tagged, and tagged-to-untagged. As shown in Figure 5.10 and Table B.8, we compare kernel forwarding performance with three `xdp-forward` variants: unmodified `xdp-forward`, and our two extension solutions.

For non-VLAN traffic, unmodified `xdp-forward` outperforms kernel forwarding, as already presented. However, when handling VLAN-tagged traffic, the unmodified version performs slower than kernel forwarding since it must pass packets to the kernel stack for VLAN handling.

Our VLAN extensions successfully address this limitation, restoring `xdp-forward`'s performance advantage across all VLAN traffic patterns. The kernel patch solution generally achieves better results than the userspace implementation. The performance overhead compared to non-VLAN traffic is approximately 0-2% for the kernel patch solution and 4-8% for the userspace implementation.

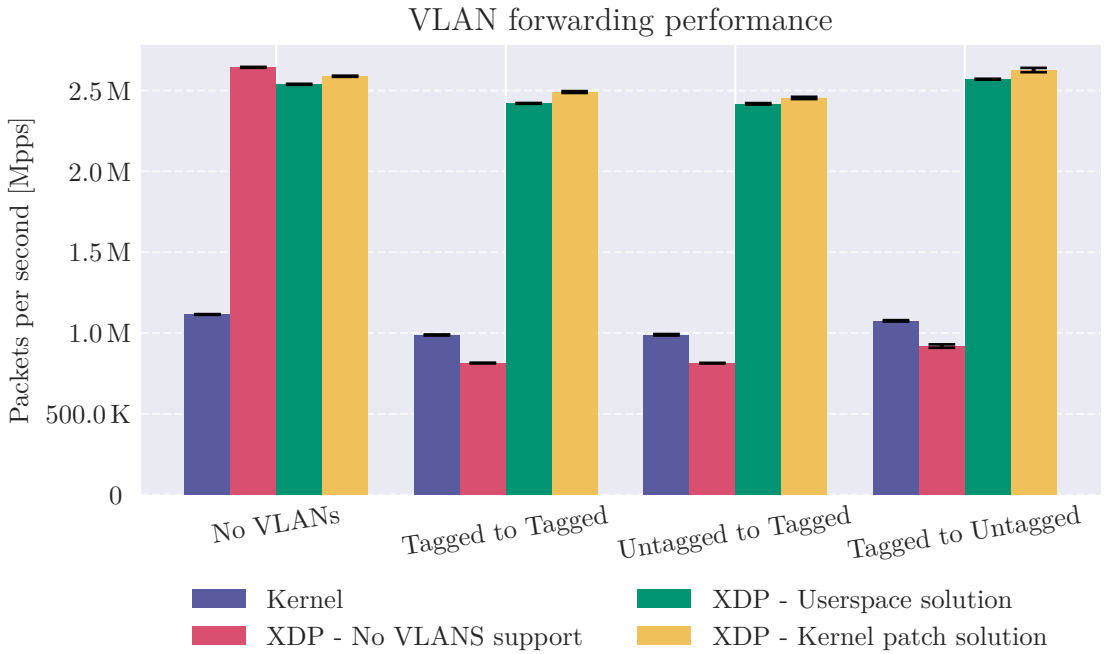

Figure 5.10: Performance comparison of different forwarding implementations across various VLAN traffic patterns. The figure shows packet forwarding rates for standard kernel forwarding, unmodified `xdp-forward`, and our two VLAN extension implementations (userspace and kernel patch). For non-VLAN traffic, unmodified `xdp-forward` demonstrates superior performance, but it falls below kernel forwarding performance for VLAN-tagged packets. Both of our VLAN extension implementations restore `xdp-forward`'s performance advantage, with the kernel patch solution achieving slightly better results than the userspace implementation.

5.4.7 Call Graph Analysis

For the final component of our performance analysis, we examined the execution profile of `xdp-forward` using Linux perf tools. Figure 5.11 presents an aggregated flame graph captured while `xdp-forward` processed 22 parallel streams on 22 CPUs with the mlx5 driver. For this test, interrupt handling for both ingress and egress NICs was pinned to the same CPUs.

The flame graph reveals multiple performance hotspots. The most significant time consumer is the FIB lookup process, which accounts for approximately 26% of execution time. This includes all the functions called by underlying functions of `bpf_xdp_fib_lookup`. This makes routing table lookups a primary candidate for further optimization efforts.

The second major time consumer is packet redirection, with `xdp_do_redirect` alone consuming 21% of CPU time. This function handles the process of enqueuing packets to the egress device's transmission queue. The actual packet transmission (visible in functions like `mlx5e_xdp_xmit` and `mlx5e_xmit_xdp_frame`) accounts for approximately 6.2% of execution time. Another significant processing cost (about 10%) is associated with memory management, particularly returning DMA pages from the egress NIC back to the ingress device's page pools. This is visible as calling functions like `xdp_return_frame_bulk` and others.

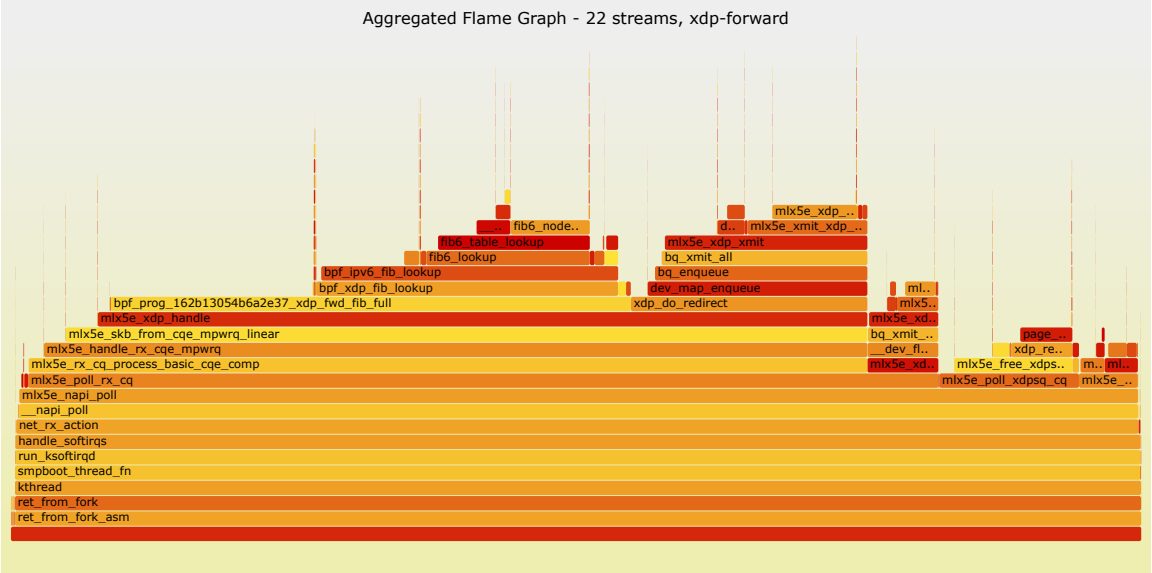

Figure 5.11: Aggregated flame graph of `xdp-forward` processing 22 streams on 22 CPUs using the mlx5 driver. The call stacks are aggregated on function `ret_from_fork_asm` to provide a unified view. The most significant time consumer is the actual FIB lookup.

5.4.8 Summary

The performance tests we ran comparing `xdp-forward` to standard kernel forwarding showed several important results.

When testing with a single stream, `xdp-forward` was faster than kernel forwarding by 1.3× to 3.8× on all hardware we tested, with the Intel and ice machine configuration reaching the highest speed at 5.0 *Mpps*. This performance advantage continued when we added more CPUs. Even with 22 parallel streams, `xdp-forward` processed 47.5 *Mpps* compared to kernel's 15.2 *Mpps*, maintaining a 3.1× advantage. We also noticed that kernel forwarding was more stable in its drop rates while `xdp-forward` showed more variation. This makes sense because the kernel has built-in buffering at various stages of networking stack that helps smooth out processing, while XDP processes packets immediately with minimal buffering.

CPU usage testing showed that `xdp-forward` is more efficient, processing about 29 000 packets per CPU percentage point compared to kernel's 11 800 packets. We also found that as we added more CPU cores, `xdp-forward` kept its per-core CPU usage steady, but kernel forwarding needed more and more CPU per core, reaching over 20% extra utilization at 22 cores when forwarding packets at zero drop pace.

When we tested different packet sizes, we saw that kernel forwarding processed about the same number of packets regardless of size, but `xdp-forward` worked best with medium to large sized packets.

Our VLAN extension for `xdp-forward` successfully fixed its inability to handle tagged traffic efficiently. The kernel patch solution we developed added only 0-2% overhead compared to non-VLAN traffic.

Looking at the call graph, we identified that routing table lookups (26%) and packet redirection (21%) took most of the processing time in `xdp-forward`, showing us where to focus for future improvements.

Chapter 6

Future Work

This section discusses possible areas of future work that could extend the functionality and performance of XDP-based packet forwarding.

Hardware Offloading

When discussing XDP program execution, it's important to distinguish between native mode and hardware offloading. The native mode executes the BPF/XDP program on the host CPU within the driver context, while hardware offloading refers to complete execution of the BPF program within the NIC hardware with no CPU involvement [30].

As of writing this thesis, full hardware offloading for XDP is supported only by Netronome Agilio SmartNICs [16]. While other offloading technologies exist, such as TC (Traffic Control) rule offloading supported by various NICs, offloading entire BPF programs represents a significantly higher level of complexity. TC offloading typically supports a limited set of match-action operations, whereas XDP programs can implement arbitrary packet processing logic within the constraints of the verifier.

Hardware offloading offers significant performance benefits[1], but it has important limitations for complex forwarding programs like `xdp-forward`. The `bpf_fib_lookup` helper function, central to routing functionality, is not supported on Netronome NICs [31]. This creates a significant problem for offloading `xdp-forward`. For the NIC to perform routing lookups, it would need to maintain a synchronized copy of the kernel's FIB, or query the kernel for lookup results, introducing latency.

A potential approach could implement map-based route caching, where a userspace daemon populates discovered routes into a BPF map accessible by the offloaded XDP program, with a fallback path to the kernel when no cached route exists.

Future Kernel Features Integration

Several kernel features under development could enhance `xdp-forward`'s functionality. XDP queuing [3] would address the packet queuing limitations discussed in Section 3.4, enabling better handling of traffic between interfaces with different capacities. Integration with netfilter functionality (Section 3.1) through new BPF helper functions could provide stateful packet filtering capabilities that are currently missing from XDP-based solutions.

[1] [30] states that for packet load-balancing, performance is up to 20× higher than with native XDP.

Chapter 7

Conclusion

This thesis explored packet forwarding acceleration in the Linux kernel, focusing primarily on the eXpress Data Path (XDP) and the `xdp-forward` utility. The work aimed to identify performance limitations of traditional kernel networking and evaluate XDP-based alternatives.

Our analysis revealed several features present in the standard kernel networking stack but absent in `xdp-forward`. These included proper VLAN support, Netfilter functionality for firewalling and connection tracking, and packet queueing mechanisms. The architectural differences between these approaches directly impact their respective feature sets.

A practical contribution of this work was the implementation of VLAN support for `xdp-forward`. We developed two approaches: one requiring a patched kernel that modifies the `bpf_fib_lookup` helper function behavior, and another using userspace-managed mapping compatible with unmodified kernels. The changes have been submitted as a merge request to the upstream `xdp-tools` project. Performance testing of our VLAN implementation showed reasonable overhead compared to baseline `xdp-forward` performance. The added VLAN processing created approximately 4% overhead for IPv6 traffic and 10% for IPv4 traffic.

Broader performance comparisons between `xdp-forward` and traditional kernel forwarding identified the Intel's `ice` driver as particularly effective for XDP workloads. With this driver, `xdp-forward` achieved packet forwarding rates up to 4× faster than the kernel's forwarding path. We noted `xdp-forward` showed more significant performance degradation for IPv6 compared to IPv4 (7% to 28%) than the kernel forwarding path (1% to 10%).

Multi-CPU forwarding tests revealed scaling limitations in the kernel that capped performance at approximately 10 Mpps. Further investigation allowed us to identify and resolve this bottleneck, enabling a higher forwarding rate. We developed a custom method for analyzing packet drop rates. The packet size impact analysis showed different behaviors between forwarding planes. Kernel forwarding maintained relatively consistent rates regardless of packet size, while XDP-based forwarding showed a stronger correlation between packet size and performance.

Finally, we explored hardware NIC offloading as a future direction. While some sources suggest potential 20× performance improvements, offloading `xdp-forward` presents challenges due to its complexity exceeding current hardware offload capabilities. Comprehensive XDP offload support currently exists only for Netronome NICs, limiting broader adoption.

Bibliography

[1] Ahern, D. *RSS/RPS + locking qdisc*. August 2020. Available at: https://people.kernel.org/dsahern/rss-rps-locking-qdisc. [Accessed 13-05-2025].

[2] Ahern, D. *Bpf_fib_lookup VLAN*. Kernel VGER, 16. April 2021. Available at: https://lore.kernel.org/xdp-newbies/9e19881a-1f62-410f-8dec-0eff0c7ea03b@gmail.com/.

[3] Alfredsson, F.; Hurtig, P.; Brunstrom, A.; Høiland Jørgensen, T. and Brouer, J. D. XDQ: Enhancing XDP with Queuing and Packet Scheduling. In: *2024 27th Conference on Innovation in Clouds, Internet and Networks (ICIN)*. 2024, p. 52–56. ISBN 979-8-3503-9376-7.

[4] Apalodimas, I. and Bianconi, L. *XDP and page_pool API*. February 2020. Available at: https://archive.fosdem.org/2020/schedule/event/xdp_and_page_pool_api/. FOSDEM'20.

[5] Ayuso, P. N. Netfilter's Connection Tracking System. *Linux Journal*. Linux Journal, LLC, November 2006, vol. 31, no. 3, p. 34–39.

[6] Bernat, V. *IPv4 route lookup on Linux*. 21. June 2017. Available at: https://vincent.bernat.ch/en/blog/2017-ipv4-route-lookup-linux. [Accessed 04-02-2025].

[7] Bertin, G. XDP in practice: integrating XDP into our DDoS mitigation pipeline. In: Cloudflare. *Technical Conference on Linux Networking (NetDev 2.1)*. The Netdev Society, 2017. Available at: https://netdevconf.info/2.1/papers/Gilberto_Bertin_XDP_in_practice.pdf.

[8] Borkmann, D. *XDP: Failing to attach XDP program to a tagged VLAN interface*. GitHub, 24. May 2023. Available at: https://github.com/cilium/cilium/issues/24768#issuecomment-1561224279. [Accessed 13-12-2024].

[9] Bovet, D. P. and Cesati, M. *Understanding the Linux Kernel: from I/O ports to process management*. 1stth ed. „ O'Reilly Media, Inc.“, 2005. ISBN 0-596-00002-2.

[10] Bradner, S. and McQuaid, J. *Benchmarking Methodology for Network Interconnect Devices* RFC 2544. 2544. RFC Editor, march 1999. Available at: https://www.rfc-editor.org/info/rfc2544.

[11] Brouer, J. D. *Introduction to: XDP and BPF building blocks*. October 2019. Available at: https://people.netfilter.org/hawk/presentations/ebplane2019/xdp-bpf-building-blocks.pdf. Ebplane by Juniper.

[12] Chaignon, P. *The Cost of BPF Tail Calls*. Mar 2021. Available at: https://pchaigno.github.io/ebpf/2021/03/22/cost-bpf-tail-calls.html. [Accessed 20-12-2024].

[13] Cilium Authors. *BPF Architecture – Cilium 1.17.0-Dev Documentation*. Cilium, Dec 2024. Available at: https://docs.cilium.io/en/latest/reference-guides/bpf/architecture/. [Accessed 20-12-2024].

[14] Damato, J. *Monitoring and Tuning the Linux Networking Stack: Receiving Data | Packagecloud Blog — blog.packagecloud.io*. packagecloud.io, June 2016. Available at: https://blog.packagecloud.io/monitoring-tuning-linux-networking-stack-receiving-data/. [Accessed 03-11-2025].

[15] eBPF.io documentation authors. *Helper functions – eBPF Docs*. 2024. Available at: https://docs.ebpf.io/linux/helper-function/. [Accessed 05-01-2025].

[16] eBPF.io documentation authors. *Program Type 'BPF_PROG_TYPE_XDP' – eBPF Docs*. eBPF.io, 2024. Available at: https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_XDP/#xdp_flags_hw_mode. [Accessed 05-01-2025].

[17] Emmerich, P.; Pudelko, M.; Bauer, S. and Carle, G. User Space Network Drivers. In: *Proceedings of the Applied Networking Research Workshop*. New York, NY, USA: ACM, 2018-07-16, p. 91–93. ISBN 9781450355858. Available at: https://dl.acm.org/doi/10.1145/3232755.3232767.

[18] Frazier, H.; Van Doorn, S.; Hays, R.; Muller, S.; Tolley, B. et al. *IEEE 802.3ad Link Aggregation (LAG): what it is, and what it is not*. Ottawa, Canada: IEEE 802.3 Higher Speed Study Group, April 2007. Available at: https://www.ieee802.org/3/hssg/public/apr07/frazier_01_0407.pdf.

[19] Herbert, T. and Bruijn, W. de. *Scaling in the Linux Networking Stack* Linux Kernel Documentation. Aug 2018. Available at: https://www.kernel.org/doc/Documentation/networking/scaling.txt. [Accessed 18-10-2025].

[20] Høiland Jørgensen, T. and Brouer, J. D. *XDP Programming Hands-On Tutorial*. GitHub, 2019. Available at: https://github.com/xdp-project/xdp-tutorial. [Accessed 04-09-2025].

[21] Høiland Jørgensen, T.; Brouer, J. D.; Borkmann, D.; Fastabend, J.; Herbert, T. et al. 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*. New York, NY, USA: Association for Computing Machinery, 2018, p. 54–66. CoNEXT '18. ISBN 9781450360807. Available at: https://doi.org/10.1145/3281411.3281443.

[22] Horáček, v. *XDP test suite for Linux kernel.* 2020. Available at: https://is.muni.cz/th/tq8qf/.

[23] IEEE. IEEE Standard for Local and metropolitan area networks–Bridges and Bridged Networks. *IEEE Std 802.1Q-2014 (Revision of IEEE Std 802.1Q-2011)*, 2014.

[24] Kerling, P. *Design, implementation, and test of a tri-mode Ethernet MAC on an FPGA*. Ilmenau, 2019. Dissertation. Available at: https://www.db-thueringen.de/receive/dbt_mods_00038245. Bachelorarbeit, Technische Universität Ilmenau, 2015.

[25] Kerrisk, M. *Bpf-helpers(7) – Linux manual page.* January 2024. Available at: https://man7.org/linux/man-pages/man7/bpf-helpers.7.html. [Accessed 12-11-2024].

[26] Maki, J. *XDP bonding support.* Kernel VGER, 30. July 2021. Available at: https://lore.kernel.org/all/20210730061822.6600-1-joamaki@gmail.com/.

[27] Matvienko, A. *Latest Trex on ubuntu 24.04 with python 12?* 2024. Available at: https://groups.google.com/g/trex-tgn/c/iz06h7vkdgA/m/u98a2f5pAgAJ. [Accessed 12-12-2024].

[28] Matz, M.; Hubička, J.; Jaeger, A. and Mitchell, M. *System V Application Binary Interface: AMD64 Architecture Processor Supplement.* SUSE, July 2012. Available at: https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf.

[29] Miller, D. S.; Henderson, R. and Jelinek, J. *Dynamic DMA mapping Guide* Linux Kernel Documentation. Technical Documentation. Linux Kernel, April 2024. Available at: https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt. [Accessed 03-10-2024].

[30] Monnet, Q. *The Challenges of XDP Hardware Offload.* February 2018. Available at: https://archive.fosdem.org/2018/schedule/event/xdp/. FOSDEM'18.

[31] Netronome. *EBPF Offload Getting Started Guide* online. Aug 2018. Available at: https://help.netronome.com/support/solutions/articles/36000050009-agilio-ebpf-2-0-6-extended-berkeley-packet-filter. Revision 1.2.

[32] Nguyen, T. L.; Silbermann, M. and Wilcox, M. *The MSI Driver Guide HOWTO.* 6.12th ed. Intel Corporation, 2008. Available at: https://docs.kernel.org/PCI/msi-howto.html. [Accessed 14-10-2025].

[33] Olsson, R. Pktgen the linux packet generator. In: *Proceedings of the Linux Symposium, Ottawa, Canada.* 2005, vol. 2. Available at: https://www.kernel.org/doc/ols/2005/ols2005v2-pages-19-32.pdf.

[34] Pedroni, V. A. *Circuit design with VHDL.* 3rdth ed. MIT press, 2020. ISBN 0262042649.

[35] Pollitt, A. *Linux Conntrack: Why it breaks down and avoiding the problem.* Tigera, April 2019. Available at: https://www.tigera.io/blog/when-linux-conntrack-is-no-longer-your-friend/. [Accessed 03-02-2025].

[36] Rice, L. *Learning EBPF*. 1stth ed. O'Reilly Media, 2023. ISBN 9781098135096.

[37] Richards, M. *Extreme HTTP Performance Tuning: 1.2M API req/s on a 4 vCPU Instance*. May 2021. Available at: https://talawah.io/blog/extreme-http-performance-tuning-one-point-two-million/. [Accessed 14-01-2025].

[38] Rosen, R. *Linux Kernel Networking: Implementation and Theory*. 1stth ed. USA: Apress, 2013. ISBN 143026196X.

[39] Salzman, P. J.; Burian, M.; Pomerantz, O. et al. *The linux kernel module programming guide*. ver, 2001.

[40] Siemon, D. Queueing in the Linux network stack. *Linux Journal*. Belltown Media Houston, TX, 2013, vol. 2013, no. 231, p. 2.

[41] Stender, A. *Routing Decisions in the Linux Kernel – Part 1: Lookup and packet flow*. 04. July 2022. Available at: https://thermalcircle.de/doku.php?id=blog:linux:routing_decisions_in_the_linux_kernel_1_lookup_packet_flow. [Accessed 04-10-2024].

[42] Tanenbaum, A. and Bos, H. *Modern Operating Systems*. 4thth ed. Pearson, 2014. ISBN 013359162X.

[43] The kernel development community. *Network Devices, the Kernel, and You! The Linux Kernel documentation*. Available at: https://docs.kernel.org/networking/netdevices.html. [Accessed 10-10-2024].

[44] Torvalds, L. et al. *Linux Kernel Version 6.12* https://www.kernel.org. v6.12. 2024. Available at: https://elixir.bootlin.com/linux/v6.12/source. Released Nov 2024.

[45] TRex authors. *Mellanox support*. 2024. Available at: https://trex-tgn.cisco.com/trex/doc/trex_appendix_mellanox.html. [Accessed 01-05-2025].

Appendix A

Tuning kernel forwarding performance beyond 10 Mpps

During work on Section 5.4.1 we found that with regular kernel forwarding we cannot forward more than roughly 10 M packets per second when using multiple CPUs. However, we can easily reach more than 40 Mpps with `xdp-forward` forwarding. Therefore, we decided to investigate further what is wrong.

By forwarding more than 4× more with `xdp-forward` we can confidently say that the limit of our NIC is much higher than 10 $Mpps$. As we mentioned, all of our tests fully utilize the CPU and so the measured performance is bottlenecked by it.

Therefore, we ran our test with `perf`[1] running in the background with the following command:

```
perf record -a -g -C <list of 22 CPUs used in test> -o mlx5_22cpus -F 4000
--buildid-all --call-graph dwarf,16384 sleep 60
```

The resulting flamegraph[2] can be seen in Figure A.1. We can see, that most of the time, CPUs are running function `__dev_queue_xmit` which is output's interface transmit function. Most of the time spent in this function, CPUs spent in function `_raw_spin_lock` which is a busy waiting for queuing discipline lock to be released by some other transmitting CPU.

Our issue is similar to the problem that David Ahern faced and described in [1] and the problem that was described in [37]. Based on this and consultation with the supervisor, we tried to use non-locking queue such as `noqueue`. This significantly helped with scaling as we can see in Figure A.4. Apart from better performance, in Figure A.2 we can see that transmission is now heavily affected by `netfilter` subsystem rather than `_raw_spin_lock`.

Unfortunately, since `netfilter` subsystem is implemented as part of SELinux, which has their hooking functions hardcoded in the kernel (due to security reasons), in order to completely disable `netfilter` subsystem, we had to recompile the kernel without `netfilter` support[3]. As we can see on Figure A.4 this slightly helped.

[1] https://perfwiki.github.io/main/

[2] Flamegraphs are hierarchical visualizations of profiling data that display call stacks as colored bars whose width represents CPU time consumption, making performance bottlenecks immediately visible in code execution paths.

[3] This can be done by removing `CONFIG_NETFILTER=y` from compilation configuration file.

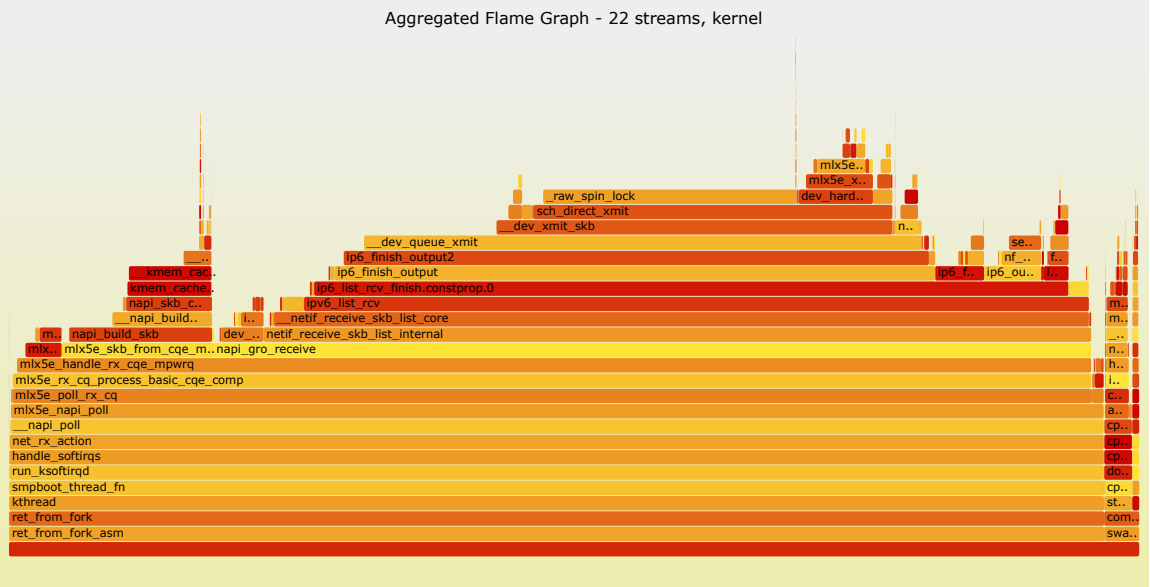

Figure A.1: The figure presents aggregated flamegraph on function `ret_from_fork_asm` of forwarding machine, that used 22 CPUs for forwarding traffic. We can see, that the most limiting factor for forwarding is in this case `__dev_queue_xmit` function, which we tracked down to be a queuing discipline lock. The machine was using default queuing discipline – `fq_codel`. The function consumed 49 % of total computation time.

The queueing discpline as well as `netfilter` subsystem are completely bypassed when running `xdp-forward`. Therefore, we used modified kernel and `noqueue` for measuring multi-cpu performance in Section 5.4.1.

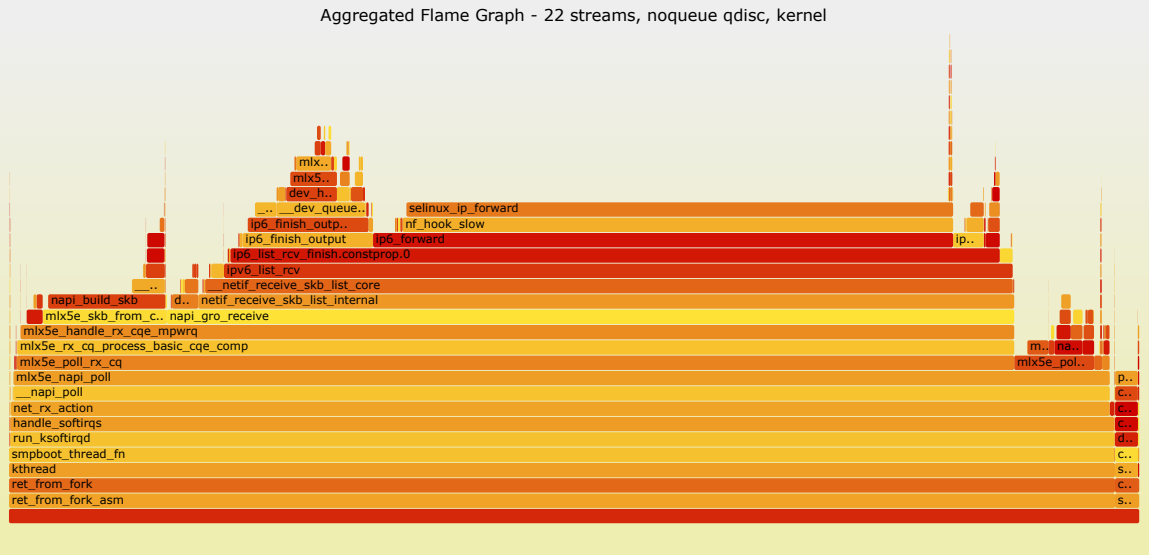

Figure A.2: The figure presents aggregated flamegraph on function `ret_from_fork_asm` of forwarding machine that used non-locking queuing discipline mechanism `noqueue`. Flamegraph revealed another function being hotspot, function `selinux_ip_forward` which handles `POSTROUTING` hooks in `netfilter` subsystem. The machine spent roughly 48 % of total computation time by running this function.

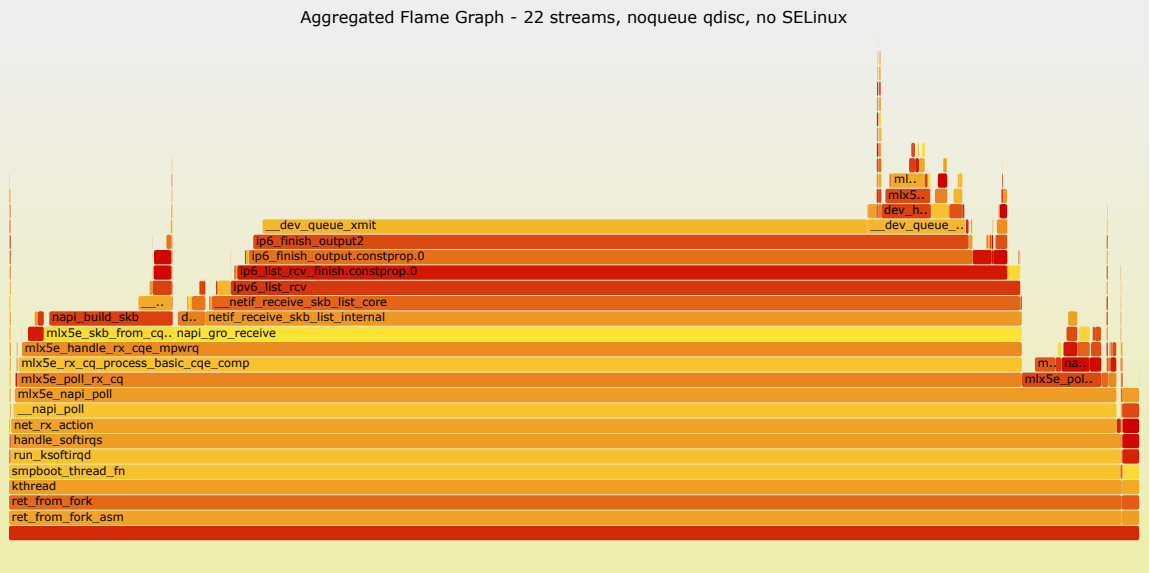

Figure A.3: This figure presents aggregated flamegraph on function `ret_from_fork_asm` of forwarding machine with `noqueue` queuing discipline and kernel compiled without `netfilter` subsystem. We can see, the lock limiting the performance is not present anymore, as well as there is no `netfilter` function that would slow down the processing.

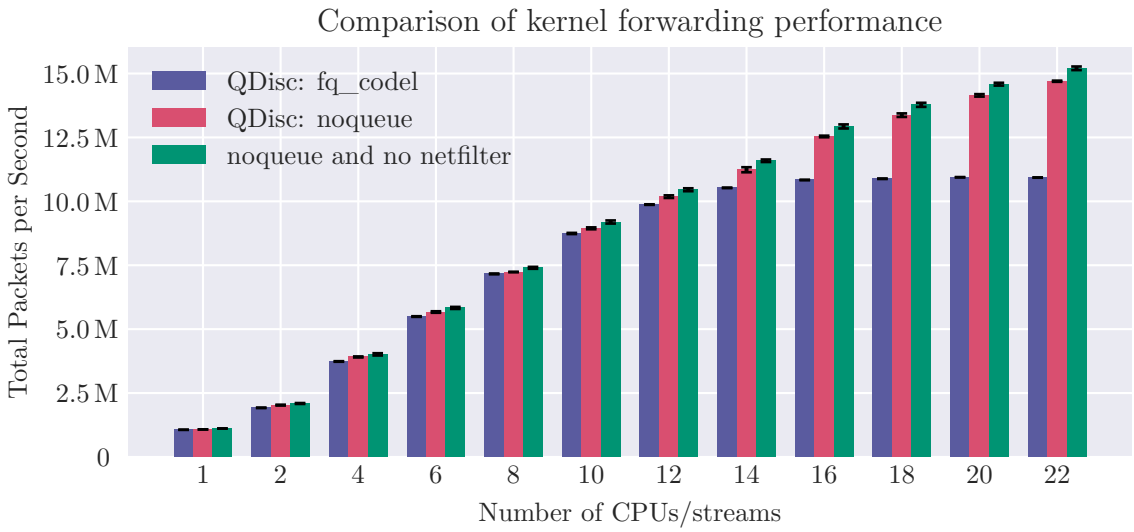

Figure A.4: Comparison of CPU scaling behaviour for various queueing disciplines and `netfilter` configuration.

CPUs	FQ-CoDel Packets/s	FQ-CoDel Multiplier	`noqueue` Packets/s	`noqueue` Multiplier	`noqueue`, No `netfilter` Packets/s	`noqueue`, No `netfilter` Multiplier
1	$1.1\,M \pm 1.6\,K$		$1.1\,M\,(+1.3\,\%) \pm 2.5\,K$		$1.1\,M\,(+4.8\,\%) \pm 2.1\,K$	
2	$1.9\,M \pm 9.1\,K$	1.81×	$2.0\,M \pm 16.2\,K$	1.88×	$2.1\,M \pm 17.6\,K$	1.88×
4	$3.7\,M \pm 10.4\,K$	3.51×	$3.9\,M \pm 20.2\,K$	3.63×	$4.0\,M \pm 44.2\,K$	3.60×
6	$5.5\,M \pm 11.9\,K$	5.16×	$5.7\,M \pm 29.1\,K$	5.26×	$5.8\,M \pm 39.7\,K$	5.23×
8	$7.2\,M \pm 15.2\,K$	6.73×	$7.2\,M \pm 8.0\,K$	6.71×	$7.4\,M \pm 37.4\,K$	6.64×
10	$8.7\,M \pm 22.6\,K$	8.22×	$8.9\,M \pm 37.0\,K$	8.30×	$9.2\,M \pm 60.5\,K$	8.24×
12	$9.9\,M \pm 10.2\,K$	9.28×	$10.2\,M \pm 50.8\,K$	9.44×	$10.5\,M \pm 56.5\,K$	9.38×
14	$10.5\,M \pm 9.4\,K$	9.89×	$11.2\,M \pm 98.1\,K$	10.42×	$11.6\,M \pm 48.7\,K$	10.39×
16	$10.8\,M \pm 9.0\,K$	10.18×	$12.5\,M \pm 30.5\,K$	11.63×	$12.9\,M \pm 79.0\,K$	11.60×
18	$10.9\,M \pm 9.4\,K$	10.23×	$13.4\,M \pm 69.0\,K$	12.40×	$13.8\,M \pm 78.9\,K$	12.36×
20	$10.9\,M \pm 7.1\,K$	10.28×	$14.1\,M \pm 47.8\,K$	13.12×	$14.6\,M \pm 54.6\,K$	13.08×
22	$10.9\,M \pm 5.7\,K$	10.27×	$14.7\,M \pm 22.4\,K$	13.63×	$15.2\,M \pm 69.9\,K$	13.64×

Table A.1: Comparison of various queuing discipline and enablement of `netfilter` for multiple CPUs. The first row presents results for single CPU and we can see that using `noqueue` with `netfilter` enabled increases forwarder packet by 1.3 % compared to `fq_codel` and it increases amount of forwarder packets by 4.8 % for setup with `noqueue` with no `netfilter` running on system. The numbers are based on IPv6 measurements.

Appendix B

Detailed Measurements Results

Basic Tests Results

				XDP				Kernel	
		IPv4		IPv6		IPv4		IPv6	
CPU	NIC	Mean	StdDev	Mean	StdDev	Mean	StdDev	Mean	StdDev
Intel1	ice	5.0 M	9.5 K	4.1 M (−17.8%)	59.7 K	1.2 M	20.0 K	1.1 M (−9.9%)	75.6 K
	mlx5	3.2 M	35.3 K	2.6 M (−16.8%)	3.0 K	1.1 M	24.6 K	1.1 M (−3.1%)	21.4 K
AMD	bnxt_en	2.5 M	35.9 K	1.8 M (−28.4%)	17.4 K	821.5 K	4.9 K	748.6 K (−8.9%)	4.4 K
ARM	ice	1.8 M	20.9 K	1.6 M (−7.0%)	1.1 K	929.5 K	23.2 K	917.5 K (−1.3%)	7.7 K
	mlx5	1.7 M	5.6 K	1.5 M (−11.9%)	3.8 K	814.6 K	30.7 K	782.7 K (−3.9%)	1.7 K
Intel2	sfc	1.4 M	5.3 K	1.2 M (−14.8%)	4.3 K	1.1 M	8.1 K	1.1 M (−1.9%)	3.6 K

Table B.1: Comparison of `xdp-forward` and kernel forwarding performance across various CPU architectures and NIC drivers. Values show mean forwarding rates (Mpps) with standard deviations for IPv4 and IPv6 traffic. Percentages in parentheses indicate IPv6 performance relative to IPv4. XDP consistently outperforms kernel forwarding by 1.3× to 3.8× across all configurations, with Intel Sapphire and ice achieving the highest absolute performance. IPv6 processing shows 7-28% lower performance than IPv4 for XDP forwarding.

IP			Kernel				XDP	
	IPv4		IPv6		IPv4		IPv6	
Metric	Mean, StdDev	Multiplier	Mean, StdDev	Multiplier	Mean, StdDev	Multiplier	Mean, StdDev	Multiplier
CPUs								
1	$1.1\,M \pm 24.1\,K$	$1.00\times$	$1.1\,M \pm 21.0\,K$	$0.98\times$	$3.2\,M \pm 33.5\,K$	$1.00\times$	$2.6\,M \pm 2.8\,K$	$1.00\times$
2	$2.1\,M \pm 20.3\,K$	$1.86\times$	$2.1\,M \pm 17.6\,K$	$1.88\times$	$6.1\,M \pm 21.3\,K$	$1.88\times$	$5.2\,M \pm 13.6\,K$	$1.95\times$
4	$4.0\,M \pm 44.0\,K$	$3.57\times$	$4.0\,M \pm 44.2\,K$	$3.60\times$	$11.9\,M \pm 35.8\,K$	$3.69\times$	$10.3\,M \pm 41.3\,K$	$3.89\times$
6	$5.8\,M \pm 43.2\,K$	$5.19\times$	$5.8\,M \pm 39.7\,K$	$5.23\times$	$17.1\,M \pm 84.8\,K$	$5.30\times$	$15.1\,M \pm 21.5\,K$	$5.69\times$
8	$7.4\,M \pm 34.0\,K$	$6.59\times$	$7.4\,M \pm 37.4\,K$	$6.63\times$	$22.8\,M \pm 116.3\,K$	$7.08\times$	$20.1\,M \pm 70.7\,K$	$7.60\times$
10	$9.2\,M \pm 43.8\,K$	$8.22\times$	$9.2\,M \pm 60.5\,K$	$8.24\times$	$27.8\,M \pm 208.5\,K$	$8.63\times$	$24.7\,M \pm 108.7\,K$	$9.33\times$
12	$10.5\,M \pm 64.6\,K$	$9.32\times$	$10.5\,M \pm 56.5\,K$	$9.37\times$	$31.0\,M \pm 182.1\,K$	$9.61\times$	$29.3\,M \pm 126.7\,K$	$11.05\times$
14	$11.6\,M \pm 74.1\,K$	$10.38\times$	$11.6\,M \pm 48.7\,K$	$10.39\times$	$35.3\,M \pm 336.8\,K$	$10.94\times$	$33.0\,M \pm 328.1\,K$	$12.48\times$
16	$12.9\,M \pm 55.6\,K$	$11.53\times$	$12.9\,M \pm 79.0\,K$	$11.59\times$	$39.8\,M \pm 530.1\,K$	$12.34\times$	$37.2\,M \pm 375.7\,K$	$14.06\times$
18	$13.8\,M \pm 67.4\,K$	$12.28\times$	$13.8\,M \pm 78.9\,K$	$12.35\times$	$44.1\,M \pm 1.0\,M$	$13.69\times$	$38.6\,M \pm 335.0\,K$	$14.58\times$
20	$14.6\,M \pm 92.4\,K$	$12.98\times$	$14.6\,M \pm 54.6\,K$	$13.07\times$	$46.6\,M \pm 883.7\,K$	$14.44\times$	$41.5\,M \pm 525.0\,K$	$15.66\times$
22	$15.2\,M \pm 88.5\,K$	$13.53\times$	$15.2\,M \pm 69.9\,K$	$13.63\times$	$47.5\,M \pm 296.6\,K$	$14.74\times$	$44.0\,M \pm 320.1\,K$	$16.60\times$

Table B.2: Multi-stream forwarding performance comparison between kernel and `xdp-forward` implementation. The table presents mean forwarding rates (in Mpps) with standard deviation across different stream counts for both IPv4 and IPv6 traffic. The multiplier columns indicate the scaling factor relative to single-stream performance. Both forwarding planes demonstrate scaling as stream count increases, with `xdp-forward` maintaining a significant performance advantage over kernel forwarding across all configurations. The data shows that at maximum tested configuration (22 streams), kernel forwarding achieves a 13.5× scaling factor for IPv4, while `xdp-forward` reaches 14.7×, indicating slightly more efficient parallel processing capabilities.

Drop Rates Tests Results

	Kernel		XDP	
Desired Drop Rate [%]	Acceptable PPS	Actual Drop Rate [%]	Acceptable PPS	Actual Drop Rate [%]
0.0	$228.4\,K \pm 7.2\,K$	0.00 ± 0.00	$477.0\,K \pm 35.2\,K$	0.00 ± 0.00
0.1	$245.1\,K \pm 7.4\,K$	0.06 ± 0.03	$637.3\,K \pm 28.3\,K$	0.04 ± 0.04
0.2	$254.1\,K \pm 12.5\,K$	0.15 ± 0.03	$788.3\,K \pm 53.4\,K$	0.16 ± 0.09
0.3	$264.2\,K \pm 15.6\,K$	0.22 ± 0.06	$1.0\,M \pm 266.1\,K$	0.18 ± 0.16
0.4	$276.4\,K \pm 21.2\,K$	0.34 ± 0.09	$1.0\,M \pm 221.9\,K$	0.32 ± 0.09
0.5	$294.5\,K \pm 3.4\,K$	0.46 ± 0.04	$981.0\,K \pm 112.9\,K$	0.41 ± 0.09
0.6	$309.4\,K \pm 13.2\,K$	0.50 ± 0.11	$1.1\,M \pm 164.0\,K$	0.52 ± 0.07
0.7	$333.9\,K \pm 4.6\,K$	0.60 ± 0.12	$1.6\,M \pm 258.3\,K$	0.59 ± 0.07
0.8	$340.3\,K \pm 8.4\,K$	0.72 ± 0.11	$1.4\,M \pm 244.8\,K$	0.70 ± 0.05
0.9	$359.8\,K \pm 11.4\,K$	0.87 ± 0.03	$1.5\,M \pm 136.2\,K$	0.68 ± 0.19
1.0	$386.7\,K \pm 17.0\,K$	0.89 ± 0.06	$1.6\,M \pm 116.8\,K$	0.83 ± 0.21
1.2	$419.6\,K \pm 10.7\,K$	0.97 ± 0.25	$1.7\,M \pm 80.1\,K$	1.10 ± 0.09
1.4	$457.7\,K \pm 36.2\,K$	1.13 ± 0.39	$1.6\,M \pm 119.1\,K$	1.26 ± 0.16
1.6	$501.2\,K \pm 21.7\,K$	1.48 ± 0.12	$1.8\,M \pm 83.0\,K$	1.43 ± 0.25
1.8	$545.9\,K \pm 32.4\,K$	1.46 ± 0.46	$1.8\,M \pm 146.3\,K$	1.24 ± 0.70
2.0	$565.1\,K \pm 17.0\,K$	1.74 ± 0.14	$1.9\,M \pm 29.5\,K$	1.78 ± 0.19
2.5	$669.6\,K \pm 12.4\,K$	2.15 ± 0.19	$2.0\,M \pm 225.5\,K$	2.39 ± 0.11
3.0	$792.4\,K \pm 35.9\,K$	2.83 ± 0.12	$2.2\,M \pm 88.0\,K$	2.83 ± 0.08
5.0	$1.0\,M \pm 75.9\,K$	3.26 ± 1.19	$2.3\,M \pm 154.9\,K$	3.46 ± 1.14

Table B.3: Drop rate stability comparison between kernel and `xdp-forward` forwarding implementations. The table presents the maximum acceptable packet rates (PPS) that achieve specified target drop percentages, along with the actual measured drop rates for each configuration. Values show mean ± standard deviation across five test iterations. Both implementations show a generally predictable relationship between increasing packet rates and drop percentages.

Packet Rate For Desired Drop Rate

	Kernel						XDP					
	IPv4			IPv6			IPv4			IPv6		
	Mean	StdDev	Ratio	Mean	StdDev	Ratio	Mean	StdDev	Ratio	Mean	StdDev	Ratio
30.0 K	2.82	0.04	10 647	2.85	0.27	10 540	1.26	0.03	23 720	1.42	0.02	21 153
60.0 K	5.35	0.06	11 209	5.59	0.03	10 739	1.96	0.02	30 567	2.54	0.03	23 600
90.0 K	7.94	0.07	11 333	8.17	0.07	11 015	3.19	0.06	28 221	3.68	0.03	24 433
120.0 K	10.27	0.08	11 681	11.69	0.11	10 262	4.25	0.07	28 243	4.78	0.03	25 092
150.0 K	12.86	0.11	11 666	13.52	0.07	11 095	5.19	0.07	28 900	5.86	0.05	25 606
180.0 K	15.46	0.10	11 643	16.16	0.06	11 138	6.20	0.05	29 038	6.98	0.03	25 799
210.0 K	17.90	0.15	11 729	18.64	0.11	11 267	7.28	0.10	28 862	8.14	0.06	25 806
230.2 K	19.43	0.11	11 849	20.73	0.12	11 104						
240.0 K	20.39	0.06	11 768	21.25	0.06	11 293	8.26	0.11	29 041	9.23	0.05	26 000
270.0 K	22.96	0.08	11 762	23.85	0.09	11 319	8.90	0.66	30 325	10.22	0.06	26 408
300.0 K	25.76	0.13	11 645	26.50	0.10	11 320	10.07	0.17	29 804	11.35	0.09	26 436
330.0 K	27.85	0.14	11 849	29.10	0.08	11 339	11.00	0.07	29 998	12.44	0.07	26 519
360.0 K	30.56	0.18	11 782	31.73	0.12	11 347	11.97	0.19	30 066	13.60	0.10	26 471
390.0 K	32.98	0.24	11 827	34.34	0.13	11 358	13.30	0.14	29 333	14.55	0.13	26 809
420.0 K	35.38	0.26	11 871	37.06	0.13	11 334	14.27	0.13	29 437	15.85	0.16	26 493
450.0 K	37.83	0.23	11 894	39.33	0.18	11 443	15.26	0.14	29 488	16.91	0.12	26 615
484.5 K							16.12	0.26	30 065	18.21	0.12	26 611

Table B.4: CPU utilization metrics for kernel and `xdp-forward` implementations across increasing packet rates. For each packet rate and protocol combination, the table presents mean CPU usage percentage, standard deviation, and efficiency ratio (packets processed per percentage of CPU utilization). The data demonstrates that `xdp-forward` maintains almost 3× higher efficiency than kernel forwarding across all tested rates, processing around 29 000 packets per CPU percentage point compared to the kernel's 11 800 packets. Both implementations show slightly increased CPU usage for IPv6 traffic, though the relative efficiency advantage of `xdp-forward` remains consistent across protocols.

Zero Drop Rate CPU Utilization Scaling

				Kernel			XDP
IP Version	CPUs	Average CPU Utilization	Overhead	PPS	Average CPU Utilization	Overhead	PPS
IPv4	1	19.43 ± 0.11		$228.6\,K \pm 791$	16.12 ± 0.26		$482.2\,K \pm 2.4\,K$
	2	23.51 ± 0.03	4.08 %	$229.9\,K \pm 122$	16.66 ± 0.41	0.55 %	$483.9\,K \pm 198$
	4	24.93 ± 0.62	5.5 %	$229.9\,K \pm 105$	17.01 ± 0.27	0.89 %	$484.0\,K \pm 191$
	6	23.26 ± 0.66	3.83 %	$229.8\,K \pm 104$	17.58 ± 0.19	1.47 %	$484.1\,K \pm 159$
	8	28.71 ± 1.3	9.28 %	$229.9\,K \pm 46$	17.49 ± 0.2	1.38 %	$484.0\,K \pm 190$
	10	29.72 ± 2.08	10.3 %	$229.9\,K \pm 117$	18.22 ± 0.1	2.1 %	$484.0\,K \pm 235$
	12	29.23 ± 2.98	9.8 %	$229.9\,K \pm 125$	17.42 ± 0.1	1.31 %	$484.0\,K \pm 231$
	14	31.35 ± 2.7	11.93 %	$229.8\,K \pm 90$	17.81 ± 0.13	1.7 %	$484.0\,K \pm 208$
	16	33.27 ± 1.58	13.84 %	$229.9\,K \pm 109$	18.59 ± 0.08	2.48 %	$483.9\,K \pm 243$
	18	34.65 ± 4.76	15.22 %	$230.0\,K \pm 33$	18.02 ± 0.16	1.91 %	$483.8\,K \pm 230$
	20	36.23 ± 2.56	16.8 %	$229.9\,K \pm 108$	18.45 ± 0.11	2.33 %	$484.0\,K \pm 228$
	22	39.48 ± 2.89	20.06 %	$229.9\,K \pm 112$	18.57 ± 0.08	2.45 %	$484.0\,K \pm 239$
IPv6	1	20.73 ± 0.12		$228.4\,K \pm 993$	18.21 ± 0.12		$480.9\,K \pm 2.9\,K$
	2	23.46 ± 0.15	2.73 %	$225.4\,K \pm 2.5\,K$	18.68 ± 0.28	0.47 %	$483.9\,K \pm 188$
	4	19.17 ± 0.09	−1.56 %	$224.6\,K \pm 153$	19.13 ± 0.18	0.92 %	$483.8\,K \pm 203$
	6	24.16 ± 0.1	3.43 %	$224.6\,K \pm 83$	19.37 ± 0.26	1.16 %	$484.0\,K \pm 226$
	8	25.06 ± 0.23	4.33 %	$224.6\,K \pm 115$	19.51 ± 0.15	1.3 %	$483.8\,K \pm 170$
	10	26.31 ± 0.45	5.57 %	$224.5\,K \pm 159$	19.15 ± 0.15	0.94 %	$483.8\,K \pm 108$
	12	28.38 ± 1.57	7.65 %	$224.3\,K \pm 171$	19.97 ± 0.1	1.76 %	$484.0\,K \pm 207$
	14	30.65 ± 3.54	9.91 %	$225.4\,K \pm 3.6\,K$	19.57 ± 0.17	1.37 %	$483.8\,K \pm 205$
	16	34.69 ± 1.25	13.96 %	$224.2\,K \pm 250$	19.93 ± 0.07	1.72 %	$483.9\,K \pm 291$
	18	36.1 ± 4.54	15.37 %	$224.0\,K \pm 180$	19.98 ± 0.06	1.78 %	$483.9\,K \pm 221$
	20	41.63 ± 0.94	20.9 %	$224.0\,K \pm 144$	20.16 ± 0.08	1.96 %	$484.1\,K \pm 137$
	22	44.14 ± 2.17	23.41 %	$223.9\,K \pm 285$	20.3 ± 0.07	2.1 %	$484.1\,K \pm 242$

Table B.5: CPU utilization and throughput metrics at zero drop rates. The table presents data for IPv4 and IPv6 traffic with increasing CPU core counts, showing average CPU utilization per core, overhead compared to single-core baseline, and packets forwarded per second.

Packet Size Scaling Results

		Kernel		XDP	
Platform	Pkt Size	IPv4	IPv6	IPv4	IPv6
Intel, ice	64	$1.2\,M, \pm 20.0\,K$	$1.1\,M, \pm 75.6\,K$	$5.0\,M, \pm 9.5\,K$	$4.1\,M, \pm 59.7\,K$
	128	$1.3\,M\ (+2.1\%), \pm 22.2\,K$	$1.3\,M\ (+11.3\%), \pm 14.0\,K$	$4.2\,M\ (-15.3\%), \pm 254.9\,K$	$3.7\,M\ (-9.5\%), \pm 144.8\,K$
	256	$1.3\,M\ (+3.0\%), \pm 24.0\,K$	$1.2\,M\ (+9.1\%), \pm 20.9\,K$	$4.0\,M\ (-20.1\%), \pm 4.7\,K$	$3.7\,M\ (-10.8\%), \pm 65.9\,K$
	512	$1.3\,M\ (+3.0\%), \pm 18.7\,K$	$1.2\,M\ (+7.9\%), \pm 97.1\,K$	$4.5\,M\ (-11.1\%), \pm 191.6\,K$	$4.1\,M\ (-0.0\%), \pm 109.0\,K$
	1024	$1.3\,M\ (+2.5\%), \pm 28.9\,K$	$1.3\,M\ (+11.7\%), \pm 9.3\,K$	$2.4\,M\ (-52.1\%), \pm 257.4\,K$	$2.7\,M\ (-35.1\%), \pm 5.9\,K$
	1280	$1.3\,M\ (+3.6\%), \pm 26.5\,K$	$1.2\,M\ (+9.2\%), \pm 18.4\,K$	$2.1\,M\ (-58.3\%), \pm 212.1\,K$	$2.1\,M\ (-48.1\%), \pm 5.2\,K$
	1518	$1.0\,M\ (-18.1\%), \pm 153.3\,K$	$1.1\,M\ (-5.4\%), \pm 125.9\,K$	$1.8\,M\ (-63.3\%), \pm 195.5\,K$	$1.8\,M\ (-55.6\%), \pm 166.2\,K$
Intel, mlx5	64	$1.1\,M, \pm 24.6\,K$	$1.1\,M, \pm 21.4\,K$	$3.2\,M, \pm 35.3\,K$	$2.6\,M, \pm 3.0\,K$
	128	$1.2\,M\ (+4.8\%), \pm 19.8\,K$	$1.1\,M\ (+2.7\%), \pm 2.7\,K$	$2.9\,M\ (-10.0\%), \pm 5.4\,K$	$2.6\,M\ (-0.7\%), \pm 2.9\,K$
	256	$1.2\,M\ (+3.5\%), \pm 3.2\,K$	$1.1\,M\ (+3.2\%), \pm 14.0\,K$	$3.8\,M\ (+20.4\%), \pm 6.0\,K$	$3.4\,M\ (+28.6\%), \pm 10.9\,K$
	512	$1.2\,M\ (+4.0\%), \pm 2.6\,K$	$1.1\,M\ (+2.6\%), \pm 3.9\,K$	$3.8\,M\ (+20.3\%), \pm 5.8\,K$	$3.4\,M\ (+29.2\%), \pm 7.5\,K$
	1024	$1.2\,M\ (+3.5\%), \pm 2.2\,K$	$1.1\,M\ (+2.9\%), \pm 3.7\,K$	$3.8\,M\ (+20.2\%), \pm 6.6\,K$	$3.4\,M\ (+29.4\%), \pm 3.2\,K$
	1280	$1.2\,M\ (+3.9\%), \pm 4.1\,K$	$1.1\,M\ (+3.0\%), \pm 5.0\,K$	$3.7\,M\ (+15.8\%), \pm 20.7\,K$	$3.4\,M\ (+27.4\%), \pm 17.7\,K$
	1518	$1.2\,M\ (+4.1\%), \pm 6.4\,K$	$1.1\,M\ (+2.9\%), \pm 2.2\,K$	$3.8\,M\ (+18.0\%), \pm 5.3\,K$	$3.3\,M\ (+23.7\%), \pm 4.7\,K$

Table B.6: Packet size scaling performance for kernel and `xdp-forward` implementations on Intel platforms with ice and mlx5 drivers. The table presents forwarding rates (in Mpps) with standard deviation for both IPv4 and IPv6 traffic across various packet sizes. Values in parentheses indicate percentage change relative to the 64-byte baseline. While kernel forwarding maintains consistent packet rates regardless of size, `xdp-forward` shows significant variations, with peak performance at medium packet sizes. For larger packets (1024+ bytes), performance becomes constrained by the 25G ICE NIC's bandwidth limit, particularly for `xdp-forward`, explaining the performance decline at these sizes.

VLAN Tests Results

Protocol	Baseline	Offloads enabled	Offloads disabled
IPv4	$1.2\,M \pm 9.6\,K$	$1.0\,M\,(-11.1\%) \pm 14.4\,K$	$997.6\,K\,(-13.3\%) \pm 13.5\,K$
IPv6	$1.1\,M \pm 2.1\,K$	$988.0\,K\,(-11.4\%) \pm 3.2\,K$	$976.7\,K\,(-12.4\%) \pm 2.1\,K$

Table B.7: Performance impact of VLAN handling methods on kernel forwarding. Values in parentheses indicate percentage change relative to the baseline. Hardware offloading provides only a marginal 1-2% advantage over software processing.

	Implementation	Untagged to Untagged	Tagged to Tagged	Untagged to Tagged	Tagged to Untagged
IPv4	Kernel	$1.2\,M, \pm 9.6\,K$	$1.0\,M, \pm 14.4\,K$	$994.6\,K, \pm 3.0\,K$	$1.1\,M, \pm 5.5\,K$
	No VLANs support	$3.2\,M\,(+176\%), \pm 33.5\,K$	$934.8\,K\,(-9\%), \pm 11.6\,K$	$859.2\,K\,(-14\%), \pm 2.5\,K$	$1.0\,M\,(-6\%), \pm 36.3\,K$
	Userspace	$2.9\,M\,(+156\%), \pm 37.7\,K$	$3.0\,M\,(+192\%), \pm 3.6\,K$	$3.0\,M\,(+198\%), \pm 14.0\,K$	$3.0\,M\,(+177\%), \pm 29.4\,K$
	Kernel Patch	$3.2\,M\,(+181\%), \pm 6.0\,K$	$3.0\,M\,(+188\%), \pm 6.2\,K$	$3.5\,M\,(+248\%), \pm 12.7\,K$	$3.2\,M\,(+197\%), \pm 26.0\,K$
IPv6	Kernel	$1.1\,M, \pm 2.1\,K$	$988.0\,K, \pm 3.2\,K$	$989.4\,K, \pm 4.4\,K$	$1.1\,M, \pm 4.0\,K$
	No VLANs support	$2.6\,M\,(+137\%), \pm 2.8\,K$	$814.2\,K\,(-18\%), \pm 2.2\,K$	$813.8\,K\,(-18\%), \pm 1.7\,K$	$919.3\,K\,(-15\%), \pm 10.4\,K$
	Userspace	$2.5\,M\,(+128\%), \pm 3.1\,K$	$2.4\,M\,(+145\%), \pm 2.4\,K$	$2.4\,M\,(+144\%), \pm 4.6\,K$	$2.6\,M\,(+139\%), \pm 2.6\,K$
	Kernel Patch	$2.6\,M\,(+132\%), \pm 3.5\,K$	$2.5\,M\,(+152\%), \pm 5.6\,K$	$2.5\,M\,(+148\%), \pm 6.8\,K$	$2.6\,M\,(+144\%), \pm 13.2\,K$

Table B.8: Performance comparison of forwarding implementations across VLAN traffic patterns. Values in parentheses show percentage change relative to kernel forwarding. While unmodified `xdp-forward` outperforms kernel for non-VLAN traffic but falls behind for VLAN traffic, both VLAN extension implementations restore performance advantage, with the kernel patch solution showing slightly better results than the userspace implementation.

Appendix C

Contents of the external attachment

```
/
├── analysis/
│   ├── Results.ipynb ..................Jupiter notebook for results analysis
│   ├── results/ .......................Measurement data for thesis outcomes
│   │   ├── cpu_scaling_for_zerodrop/ . Zero drop rate tests with CPU scaling
│   │   ├── multi_stream/ ..........Multi-stream tests at maximum packet rate
│   │   │   ├── kernel/ ..............................Kernel forwarding results
│   │   │   │   ├── fq_codel/ ............................fq_codel qdisc results
│   │   │   │   ├── noqueue/ ...............noqueue qdisc with netfilter results
│   │   │   │   └── noqueue+no_nf/ ............noqueue without netfilter results
│   │   ├── ndr_pps_rate/ ............................No drop rate test results
│   │   ├── queues/ .......................RX/TX ring size impact test results
│   │   ├── ratep_scaling/ ..........Packet rate scaling results (single CPU)
│   │   ├── single_stream/ .............Single stream maximum forwarding tests
│   │   │   ├── mlx5+sapphire/ ...........Tests with MLX5 and Sapphire hardware
│   │   │   │   ├── kernel_fq_codel/ .....................fq_codel qdisc results
│   │   │   │   ├── kernel/ ..............noqueue with disabled netfilter results
│   │   │   │   └── kernel_noqueue_nf_enabled/ ..noqueue with enabled netfilter
│   │   ├── various_msg_sizes/ ...................Packet size scaling results
│   │   └── vlans/ ..........................................VLAN tests results
│   │       ├── functional/ ..............pcap files for xdp-forward extensions
│   │       ├── offloads_disabled/ ..............VLAN offloads comparison tests
│   │       └── performance/ .....................VLAN performance test results
│   └── perf/ ..................................................Flamegraphs
│       └── aggregate_cpus.py  Script used for per-CPU perf stats aggregation
├── text/ ..........................................LaTeX sources of thesis
│   └── DP.pdf .....................................Compiled thesis document
└── implementation/ ............................Source code used in thesis
    ├── xdp-tools/ .....................Modified xdp-forward implementation
    ├── lnst/ ...........................Modified LNST framework for testing
    └── kernel-vs-xdp-forwarding/ ..................Custom benchmarking tool
```