UHASSELT
KNOWLEDGE IN ACTION

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

Network Stack Optimizations for QUIC

Olaf Van Bylen

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

Prof. dr. Peter QUAX

BEGELEIDER :

De heer Joris HERBOTS

De heer Mike VANDERSANDEN

2022
2023

De transnationale Universiteit Limburg is een uniek samenwerkingsverband van twee universiteiten in twee landen: de Universiteit Hasselt en Maastricht University.

UHASSELT
KNOWLEDGE IN ACTION

www.uhasselt.be
Universiteit Hasselt
Campus Hasselt:
Martelarenlaan 42 | 3500 Hasselt
Campus Diepenbeek:
Agoralaan Gebouw D | 3590 Diepenbeek

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

Network Stack Optimizations for QUIC

Olaf Van Bylen

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

Prof. dr. Peter QUAX

BEGELEIDER :

De heer Joris HERBOTS

De heer Mike VANDERSANDEN

Universiteit Hasselt

Masterproef voorgedragen tot het behalen van de graad van master in de informatica

Network stack optimizations for QUIC

Auteur:

Olaf Van Bylen

Promotor:

Prof. Dr. Peter Quax

Begeleiders:

Drs. Joris Herbots
Drs. Mike Vandersanden

Academiejaar 2022-2023

Acknowledgements

I would like to thank my promotor Prof. Dr. Peter Quax for giving me the opportunity to write this thesis, the support, and the provided hardware. I also want to express my gratitude towards my mentors: Joris Herbots and Mike Vandersanden for their regular input, feedback, discussions on this topic, email responses during the weekend, and forwarding the relevant publications they found during their own research.

Abstract

QUIC, an alternative to TCP, is a new network protocol released in 2021 and designed to be compatible with older network infrastructure (hardware and software). Therefore most QUIC implementations also reuse the UDP implementation and other parts of the host network stack provided by the operating system. While TCP has had countless optimizations in the kernel boosting its efficiency, QUIC performance is partially held back by implementation and design choices made before its existence. In this thesis, we use an experimental approach to discover and test the main bottlenecks in network stacks for QUIC performance

To do so we start by looking into important background information: the protocols and headers that we need to implement, the packet flow through the Linux kernel network stack, the important things to consider in terms of performance and how AF_XDP, the kernel bypass solution that we will be using, works.

We use this knowledge in combination with prior research to determine good candidates for performance optimizations. To easily experiment with changes and optimizations that can be made to a network stack we implement a relatively simple network stack in user space by utilizing AF_XDP sockets. The main advantage of this choice is that the network stack can easily be modified without having the recompile the entire Linux kernel. We essentially develop a UDP socket on top of AF_XDP for which we provide an interface similar to the UDP socket interface provided by the Linux kernel. This is to show the feasibility of offering a user space network stack and the potential performance benefits it brings as an almost drop-in replacement. For our testing, we utilize lsquic to implement an HTTP/3 server and client to perform performance measurements for a real-world application: an HTTP/3 file transfer over QUIC.

By utilizing our user space network stack to experiment we do find several factors that heavily impact performance. The number of system calls, the number of data copies, the use of checksumming, and the size of buffers all impact performance. However, it is also clear that not all optimizations always provide better performance, for the Linux kernel network stack we do not measure improved performance by using hardware offloading. If we disable checksumming entirely in our user space network stack we also do not measure a significant difference unless we also increase the send buffer size. We conclude that the applied optimizations do have a significant impact on QUIC performance in our tests and we expect them to improve QUIC performance in general. However how much they improve performance might vary depending on which other bottlenecks are present, the use case and used hardware.

Nederlandstalige Samenvatting

QUIC, een alternatief voor TCP is een relatief nieuw protocol dat werd gestandardiseerd in 2021 en ontworpen is om compatibel te zijn met oudere netwerk infrastructuur. Daarom hergebruiken de meeste QUIC implementaties ook bestaande delen van de host netwerk stack (bijvoorbeeld in de Linux kernel). Terwijl er voor TCP talloze optimalisaties zijn toegepast doorheen de jaren worden de prestaties van QUIC deels tegengehouden door implementatie en design keuzes die gemaakt werden voor zijn bestaan. In deze thesis gaan we op zoek naar de voornaamste knelpunten voor de prestaties van QUIC.

Hierbij stellen we ons de onderzoeksvraag: "Welke prestatieknelpunten zijn aanwezig in network stacks die de prestaties van QUIC negatief beïnvloeden?" met als subvraag: "Kunnen we de impact van deze knelpunten verminderen of deze knelpunten compleet omzeilen en wat is de impact op een real-world test, namelijk een HTTP/3 bestandsdownload over QUIC?". We kijken hiervoor naar de prestaties gemeten als goodput en de relatieve tijd die de QUIC implementatie spendeert aan de netwerk stack en andere taken.

We kijken specifiek naar verbetering voor QUIC omdat QUIC ons een goede kandidaat lijkt door het verschil tussen zijn prestaties en die van TCP. Verder is onze hypothese dat niet alle generieke optimalisaties een even goede impact hebben op QUIC: mogelijks veroorzaken sommige optimalisaties voor een hogere throughput op UDP niveau maar ook voor een hogere latentie wat mogelijks een negatieve impact heeft op congestion control en flow control. Om de onderzoeksvraag en subvraag te beantwoorden gaan we gebruik maken van AF_XDP en kernel-bypass oplossing waarmee we een netwerk stack gaan implementeren die in user space draait en waar we makkelijk aanpassingen en optimalisaties kunnen op uitproberen. Aangezien deze methode mogelijks niet enkel in de academische wereld maar ook voor echte toepassingen een goede oplossing is, stellen we onszelf een tweede onderzoeksvraag: "Kan alle functionaliteit die aangeboden wordt door de Linux kernel netwerk stack ook geïmplementeerd worden in een netwerk stack die bovenop AF_XDP wordt geïmplementeerd?".

Achtergrondkennis

Om de onderzoeksvragen te beantwoorden bespreken we eerst de nodige achtergrondkennis. We bespreken de nodige kennis van network protocollen die we gaan gebruiken namelijk Ethernet, Internet Protocol v4 (IPv4), User Datagram Protocol (UDP) en QUIC. We bespreken de hardware waarmee we interageren en waarlangs alle data passeert, namelijk de Network Interface Controller (NIC). Hierbij leggen we de focus op de hardware offloads die deze aanbiedt, voornamelijk checksum en segmentatie offloads. Checksum offloads zorgen ervoor dat de checksum die gebruikt wordt in de IPv4 en UDP headers niet meer (volledig) door de processor moet worden berekend. Segmentatie offloads zorgen ervoor dat minder en grotere pakketten doorheen de netwerk stack kunnen vloeien en in de NIC worden opgesplitst wat toelaat minder berekeningen te doen voor dezelfde hoeveelheid data.

Verder bespreken we de packet en data flow doorheen de Linux kernel network stack en de features die deze aanbiedt. Hierbij leggen we een focus op de prestatierelevante onderdelen. Hardware interrupts worden gebruikt door de NIC om de kernel te melden dat nieuwe paketten zijn aangekomen maar deze veroorzaken een context verandering die relatief veel kost. Software

interrupts zijn vergelijkbaar maar worden door de kernel zelf gegenereerd. We zien dat NAPI wordt gebruikt om het aantal hardware en software interrupts te beperken door soms aan polling te doen met hardware interrupts van de NIC tijdelijk uitgeschakeld. Verder wanneer een applicatie die in user space draait data wil lezen of schrijven van of naar een socket moet deze communiceren met de kernel, dit gebeurt via system calls. Ook deze veroorzaken een context wisseling van user space naar kernel space welke ook kostelijk is. Bovendien wordt er steeds een geheugenkopie gemaakt tussen kernel space en user space.

We sluiten de achtergrondkennis af met de werking van AF_XDP, waar we uitleggen hoe het gedeelde geheugen tussen kernel en user space werkt, hoe de rings werken en hoe we ze kunnen gebruiken om pakketen te sturen en te ontvangen. Verder leggen we de verschillende locaties uit waar XDP pakketen kan onderscheppen: in het begin van de kernel netwerk stack of rechtstreeks in de NIC driver. Alsook hoe zero-copy ervoor zorgt dat er een geheugenkopie wordt uitgespaard.

Gerelateerd werk

In het gerelateerd werk kijken we naar andere methoden om aan kernel-bypass te doen of om netwerk stack performance te verbeteren alsook naar relevante publicaties die reeds onderzocht hebben waar prestatieknelpunten zich kunnen bevinden. Zo toont Jaeger et al. [Jae+23] aan dat netwerk stack performance een relevant topic is voor QUIC omdat in hun metingen de processor de meeste tijd spendeerd aan packet I/O, de operatie die in de netwerk stack gebeurt, bij het verzenden van data over QUIC. Andere publicaties gaan dan weer in het algemeen op zoek naar knelpunten in de prestaties van de Linux kernel network stack zoals Cai et al. [Cai+21]. Anderen lossen knelpunten op zoals Netmap [Riz12], DPDK [Foua], mTCP [Jeo+14] en Onload [Xil]. We bekijken ook kort de evolutie en prestatieverbeteringen binnen XDP alsook XDP voor Windows.

Implementatie

Voor de implementatie maken we gebruik van AF_XDP om de Linux kernel netwerk stack te omzeilen en om een simpele netwerk stack in user space te bouwen. We beginnen met te motiveren waarom er voor AF_XDP is gekozen: het is goed geïntegreerd in de Linux kernel, behoudt het gebruik van interrupts en vereist bijgevolg niet dat er altijd gepolld wordt zoals bij de meeste DPDK drivers wel het geval is. Dit heeft het voordeel dat er minder processergebruik en dus minder energieverbruik is als er weinig inkomende pakketen zijn.

We houden de API van onze implementatie gelijkend aan die van de Linux kernel zodat het makkelijk is om bestaande code om te vormen om onze netwerk stack te gebruiken. Dit passen we toe voor de voorbeeldprogramma's van lsquic, specifiek de HTTP/3 server en client. Verder bouwen we ook een testraamwerk op basis van Ansible dat ons toelaat om makkelijk testen te draaien op twee hosts en automatisch de resultaten te verzamelen naar een controlecomputer.

Ten slotte bespreken we ook hoe we functionaliteit die de kernel aanbiedt, namelijk Netfilter en het Address Resolution Protocol (ARP) kunen implementeren of vervangen door een gelijkaardig alternatief. Voor Netfilter kunnen we de functionaliteit in user space programeren of gebruik maken van eBPF. Voor ARP kunnen we de ARP pakketjes met behulp van eBPF forwarden naar de kernel of kunnen we de functionaliteit ook in user space implementeren.

Experimenten en resultaten

Bij de experimenten en resultaten bespreken we eerst de hardware die we gebruiken om te testen, vervolgens doen we enkele algemene testen om een idee te krijgen van de performantie. Daarna gaan we enkele optimalisaties uitproberen, we vinden enkele dingen die een duidelijke impact hebben op de prestaties. Een geheugenkopie vermijden, het aantal system calls verminderen door pakketjes te bundelen en slechts één system call te doen voor meerdere pakketten, de

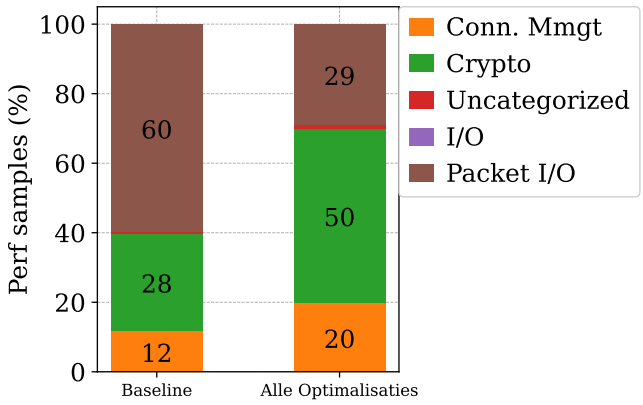

Figure 1: Relatief aantal perf samples dat gespendeerd wordt in elke categorie, onze baseline (AF_XDP_NATIVE) versus de versie met alle optimalisaties. Het aandeel van Uncategorized en I/O is zo klein dat deze onzichtbaar zijn: ze bevatten minder dan 1% van de samples en dit cijfer staat ook niet in de grafiek.

buffergroottes optimaliseren en checksumming uitzetten blijken allemaal een positief effect te hebben op de prestaties. Hoewel blijkt dat niet alle optimalisaties in alle gevallen helpen, zo hebben te kleine buffers een negatief effect op de prestaties maar dit is enkel het geval als we reeds enkele andere optimalisaties hebben toegepast. Wat erop wijst dat dit knelpunt zich enkel voordoet als de processor snel genoeg pakketten kan genereren. Verder, bij het pakketen bundelen merken we dat meer pakketjes bundelen niet altijd een voordeel is, als we lsquic toelaten tot 1024 pakketjes te bundelen zijn de prestaties slechter dan bij 32. We zien het eindresultaat versus de baseline gemeten in aantal perf samples dat de processorkern in elke categorie spendeert in Figuur 1, de netwerk stack komt overeen met de packet I/O categorie. We zien een grote reductie in relatieve tijd die aan de netwerk stack wordt gespendeerd.

Conclusie

In de conclusie besluiten we dat de gevonden factoren gecombineerd een significant effect hebben op de prestaties en dat bijgevolg de relatieve CPU tijd die gespendeerd wordt aan de netwerk stack zakt van ongeveer 60% naar 29%. Bovendien zien we dat hoewel de netwerk stack in onze implementatie eerst het onderdeel was dat veruit de meeste tijd kostte, de crypto operaties voor QUIC na onze optimalisaties het duurste worden met ongeveer 50% van het processorgebruik. Bovendien zien we dat de prestaties gemeten als goodput van 1700 Mbps voor optimalizaties naar 3348 Mbps na optimalisaties stijgen. We kunnen ook de onderzoeksvragen beantwoorden, de voornaamste knelpunten zijn: geheugenkopieën, system calls, checksumming en suboptimale buffergroottes. Verder kunnen we bevestigen dat deze knelpunten oplossen een impact heeft op de real-world performance. Daarnaast kunnen we ook de tweede onderzoeksvraag beantwoorden. Het is momenteel niet mogelijk om alle features uit de kernel netwerk stack of een gelijkwaardig alternatief in AF_XDP te gebruiken. Hardware offloads vallen volledig weg en ook voor traffic control is er momenteel geen oplossing die in alle gevallen werkt. Al bespreken we in het toekomstig werk wel manieren waarop dit in de toekomst opgelost zou kunnen worden.

Contents

1 Introduction **9**

2 Background information **11**
2.1 Overview of network stack protocols . . . 11
2.2 The Network Interface Controller (NIC) . . . 12
2.2.1 Hardware offload capabilities . . . 13
2.2.2 Hardware interrupts . . . 14
2.3 Linux kernel basics . . . 14
2.4 The basic UDP flow through the kernel . . . 16
2.4.1 Creating and binding a socket . . . 16
2.4.2 Sendmsg() . . . 16
2.4.3 Incoming packet(s) on the NIC . . . 18
2.4.4 Recvmsg() . . . 19
2.4.5 Performance considerations . . . 19
2.5 eBPF eXpress Data Path (XDP) . . . 20
2.5.1 eBPF . . . 20
2.5.2 XDP . . . 20
2.5.3 XDP address family (AF_XDP) . . . 21

3 Related work **26**
3.1 Host network stack optimizations . . . 26
3.2 Direct Cache Access (DCA) . . . 26
3.3 Alternative network stacks and kernel bypass techniques . . . 26
3.3.1 Data Plane Development Kit (DPDK) . . . 27
3.3.2 Netmap . . . 27
3.3.3 mTCP . . . 27
3.3.4 Onload . . . 28
3.4 QUIC . . . 28
3.4.1 In-kernel QUIC . . . 28
3.4.2 QUIC on top of DPDK . . . 29
3.5 XDP . . . 29
3.5.1 XDP applications . . . 29
3.5.2 Windows XDP . . . 29
3.6 Other options in the Linux kernel . . . 30

4 Implementation **31**
4.1 The choice for AF_XDP . . . 31
4.2 AF_XDP UDP socket . . . 31
4.2.1 Creating the socket . . . 32
4.2.2 Sending packets: AF_sendmsg() . . . 32
4.2.3 Receiving packets: AF_recvmsg() . . . 34
4.2.4 Batched sending of packets: AF_sendmmsg() . . . 34

4.2.5 UMEM layout and default buffer sizes . . . 35
4.2.6 Netfilter alternative . . . 35
4.2.7 ARP . . . 36
4.3 QUIC and HTTP/3 . . . 36
4.3.1 HTTP server and client . . . 36
4.4 Testing framework . . . 37

5 Experiments and results **38**
5.1 Setup . . . 38
5.2 Software . . . 38
5.3 Naming convention . . . 39
5.4 Dividing the CPU's spent time into categories . . . 40
5.5 Experiments and results . . . 41
5.5.1 General measurements . . . 41
5.5.2 Memory copies . . . 44
5.5.3 System calls . . . 44
5.5.4 Using AF_sendmmsg() . . . 46
5.5.5 Using sendmmsg() . . . 48
5.5.6 Cost of checksumming . . . 49
5.5.7 Buffer sizes . . . 50
5.5.8 Interrupts . . . 52
5.5.9 All optimizations . . . 53

6 Conclusions **55**
6.1 Future Work . . . 57
6.1.1 Traffic control . . . 57
6.1.2 Hardware offloads in AF_XDP . . . 57
6.1.3 Test on other hardware . . . 57
6.1.4 Look into NIC drivers . . . 58
6.1.5 Remove the last memory copy . . . 58

Chapter 1

Introduction

QUIC is a new protocol designed to improve on the widely used TCP/TLS stack. It was originally introduced by Google to speed up web traffic via HTTP(S) but has since evolved into a general-purpose network protocol. It aims to solve some of the biggest performance issues with TCP. It does so by using TLS 1.3 which implements 0 Round Trip Time (0RTT) for faster connection setup. Additionally, it supports connection migration, and Head-of-Line (HoL) blocking removal by supporting multiple streams at transport layer level. This results in better packet loss resilience as a lost packet in one stream does not affect the others [Marb]. QUIC runs on top of UDP which has several advantages: we can easily implement QUIC in user space on top of UDP and QUIC is backward compatible with all network hardware that supports UDP. The latter allows us to circumvent protocol ossification, the loss of flexibility, extensibility and evolvability of protocols. For example, older hardware such as firewalls might not support a new layer 4 protocol number [Mara].

Most of the current QUIC implementations that we are aware of run in user space and use the underlying (UDP) socket interface provided by the operating system kernel. This is great for compatibility and stability as, for Linux specifically, the network stack is a mature component that has been in broad use for years. So it's relatively safe to assume that there are no or little bugs and compatibility issues. Further, QUIC is still a new protocol compared to TCP and UDP and has evolved over the years before being standardized by the IETF in 2021 with RFC 9000 [IT21]. An unfinished protocol is hard to ship inside the kernel as changes to the kernel are rather slow which would slow down the evolution of QUIC. There are also many kernel versions still in use which means backward compatibility with kernels that ship with older QUIC versions would be an issue. Implementing QUIC in user space circumvents these problems as QUIC can simply be updated together with the app, which can automatically update itself. For example, you can still run the latest Chrome version including the built-in QUIC implementation on an OS running an older kernel version.

However, there are also disadvantages to this popular approach of implementing QUIC. The biggest disadvantage is the current performance compared to TCP both in terms of maximum throughput as in CPU utilization for the same throughput [Marb], due to the way the (Linux) kernel and UDP work, QUIC has to make a system call for every message it wants to transmit or receive (assuming the "standard" method of utilizing *sendmsg()* and *recvmsg()*). Further, we are only aware of a handful of popular TCP and UDP implementations: the ones provided by Linux, Windows, and MacOS. There are many more QUIC implementations and this can make it hard for all the implementations to interoperate as is evident from [See] and Marx et al. [Mar+20].

In this thesis, we will be working to answer the following main research question: "Which performance bottlenecks are present in network stacks that degrade the performance of QUIC?". We will focus on the underlying software network stack starting from UDP down to the hard-

ware. We also ask ourselves the following subquestion: "Can we decrease the impact of these bottlenecks or completely remove them and what is the effect on the performance of an HTTP/3 file download over QUIC?".

As a hypothesis for this question and subquestion, we hypothesize that user space QUIC implementations make a large number of system calls which in turn cause expensive context switches. Therefore we expect this large amount of system calls to form a bottleneck. Further, we do expect that by improving or solving this and other bottlenecks, we can improve performance. However, we do believe that testing this in reality is crucial, as we do believe not all general network stack optimizations will benefit QUIC performance. For example, an optimization that increases maximum goodput at the UDP level but also increases latency could potentially harm the goodput at the QUIC level due to flow control and congestion control not playing well with the increased latency. Additionally, a QUIC implementation also runs connection management and crypto operations which could for example impact the usage of CPU caches and how much cache is used for the network stack and some optimizations might unknowingly rely on some instructions or data being cached to achieve their performance improvement. In general, we hypothesize that there is some uncertainty about how optimizations impact specific protocols in real-world performance due to the opaqueness of modern CPUs in terms of cache management, branch predictions, etc.

To answer this research question we utilize AF_XDP, a kernel-bypass solution, to implement a user space network stack and experiment with different optimizations and modifications. As AF_XDP could potentially be a viable way to implement these improvements in real-world applications we keep the following secondary research question in mind to asses its viability: "Can all functionality offered by the Linux kernel network stack also be implemented in a network stack on top of AF_XDP?".

Chapter 2

Background information

In this chapter we explore the necessary background knowledge in order to understand our implementation while also introducing mechanisms that have a potential performance impact that we will experiment with in Chapter 5.

We begin by providing an overview of the network stack, the software layer(s) that allows applications to communicate with (applications running on) other devices and more specifically the protocols that we will have to implement in our own network stack. Next, we look at the Network Interface Controller (NIC), the device we use to transmit and receive packets. As well as a high-level view of how data passes through the Linux kernel network stack from and to the NIC. We pinpoint the regions where we can improve performance in later chapters. Interesting things include where and how copies happen, where and how we do checksumming and how the CPU core that executes certain functions is chosen in a multi-core system. While also paying attention to functionality offered by the network stack that we mimic in our own network stack such as the ARP (Address Resolution Protocol). As well as functionality offered by the network stack that we (may) lose such as Netfilter and traffic control by using a kernel bypass solution like AF_XDP.

The kernel bypass solution we will utilize to build our own network stack is AF_XDP for which we explain the basis including aspects that can have an impact on performance. We also show how one can transmit and receive data over an AF_XDP socket as we use this in our implementation and will also optimize this process to improve performance.

During this research we used Linux kernel version 6.4(.9) unless mentioned otherwise. When referring to specific source code files such as for example /net/udp.h we therefore mean the file in kernel version 6.4(.9) which is available on Github [Foud], Bootlin (which provides an easy-to-use website to view the code) [Fouc] or directly from kernel.org (using the git URL they provide) [Fouh].

2.1 Overview of network stack protocols

The Linux kernel's network stack implements layers 2 (data link), 3 (network), and 4 (transport) of the Open Systems Interconnection (OSI) model. We will be focusing on Ethernet, Internet Protocol v4 (IPv4), User Datagram Protocol (UDP) and QUIC. These are the protocols that we will be using in our implementation. Using these protocols comes down to filling the right values in their headers when sending packets or reading and checking the right values when receiving packets.

For the Ethernet header there is nothing noteworthy, as for the IPv4 and UDP headers they both contain a checksum. For the IPv4 header this is defined IETF RFC 791 [IET81], and is calculated over the entire IPv4 header. The UDP checksum is defined in IETF RFC 768

Figure 2.1: Example of a NIC, specifically the (Dell) Intel X540-T2 with a PCI-E interface and 2 Ethernet ports, image source: [Ser]

[IET80]: it is calculated over the UDP header, an additional pseudo-header, and the UDP data. In the rest of this chapter we will see where Linux calculates these checksums and in Section 4.2.2 we will describe how we do this in our implementation.

The main goal of this thesis is to improve the performance of the network stack for QUIC. While QUIC was originally designed at Google it has evolved a lot and QUIC, as defined in IETF RFC 9000 was designed by a much broader group than Google alone. QUIC, operates on top of UDP while technically also being a transport layer protocol. QUIC has been designed as a replacement for TCP and therefore the core functionality is relatively similar: it operates on data streams and it automatically retransmits data if the recipient does not confirm that it has arrived.

QUIC also has the same features as TCP to prevent it from oversaturating the network or overwhelming a slow receiver. For this congestion control and flow control are used respectively. For congestion control the algorithm can be chosen just like TCP, notable options are CUBIC [HRX08] and BBR [Car+16]. The flow control algorithm in QUIC works similarly to that of TCP.

Notable differences between QUIC and TCP are the fact that QUIC supports 0RTT connection handshake and multiple streams while also requiring encryption (which is optional for TCP). This makes encryption an inherent cost for QUIC. Additional goals for QUIC are more protocol flexibility, meaning the ability to introduce new versions of QUIC [IET]. The ability to extend QUIC using plugins was also proposed by some [Pir+20].

Further, developing and deploying changes to kernel-level networking protocols can be a time-consuming process due to the need for rigorous testing and coordination between developers. By implementing QUIC in user space, developers can iterate and update the protocol more rapidly, allowing for quicker bug fixes and the incorporation of new features. While also being able to ship applications with (new versions of) QUIC without having to wait for operating systems to implement the protocol illustrated by Google Chrome [Lan+17] which has had (google) QUIC support years before QUIC was standardized by the IETF in 2021 by RFC 9000 [IT21].

While some operating systems do have support for QUIC, notably macOS [App] and Windows [Micb] many more user space QUIC implementations exist [See] and popular applications such as Google Chrome and Mozilla Firefox also implement QUIC in user space.

2.2 The Network Interface Controller (NIC)

In this section we take a look at the Network Interface Controller as this will be the main, and only device where we will be interacting with. A physical Network Interface Controller can have multiple forms, one of which is as an expansion card with a PCI-Express interface and one or more ports. A variant of the X540-T2 we use in this thesis can be seen in Figure 2.1. It can also be integrated onto a laptop, desktop or server motherboard. In essence, its function

is taking packets from a cable (or from the air in wireless networks) and delivering them to system main memory while notifying the CPU of the new packets and vice versa where the CPU can transmit packets from main memory via the NIC to the network. In this section we will look into the hardware offloads that the NIC can provide and how it can signal the CPU that one or more new packets have arrived.

To transfer data to and from main system memory virtually all modern NICs support Direct Memory Access (DMA). This feature involves the NIC directly reading and writing data from the main memory without the CPU being involved. This of course is much more efficient compared to having to involve the CPU in this operation. In the rest of this thesis, when packets are transferred from the host to the NIC and vice versa we will assume this happens via DMA

2.2.1 Hardware offload capabilities

Hardware offloads in network interface cards refer to the ability of the NIC to perform certain networking tasks directly in hardware, offloading the burden from the host system's CPU. This capability theoretically improves overall system performance, reduces CPU utilization, and enhances network throughput. The supported hardware offload functionalities depend on the NIC, driver and operating system support.

On Linux a list of features can be shown using *"ethtool -k device_name"*. A partial output looks like this on our system:

```
$ ethtool -k enp0s25
Features for enp0s25:
rx-checksumming: on
tx-checksumming: on
     tx-checksum-ipv4: off [fixed]
     tx-checksum-ip-generic: on
     tx-checksum-ipv6: off [fixed]
     tx-checksum-fcoe-crc: off [fixed]
     tx-checksum-sctp: off [fixed]
scatter-gather: on
     tx-scatter-gather: on
     tx-scatter-gather-fraglist: off [fixed]
```

To change the offload settings we can run ethtool as root using the command: *"sudo ethtool -K device_name offload on/off"* to enable or disable an offload.

A complete list of hardware offloads supported by Linux can be found inside the kernel in /include/linux/netdev_features.h for which you can find the mappings to the names shown in ethtool in /net/ethtool/common.c. We limit ourselves to checksum offloading and segmentation offloads.

As we have seen both the IP and UDP headers contain a checksum, traditionally these checksums were always calculated in software. However today most NICs support offloading (some) checksum calculations in hardware. This both for transmitting and for receiving packets. Another hardware offload provided by many NICs is TCP Segmentation Offload (TSO) which only works for TCP. As the name suggests it offloads the segmentation of a larger packet into more small packets in the NIC. As TSO assembles the frame in the NIC it cannot work without TX checksum offload enabled. This offload is greatly beneficial for performance. There are also software alternatives to this that are more general purpose as they also work for UDP: Generic Segmentation Offload (GSO) when transmitting and Generic Receive Offload (GRO) when receiving. Even though they run in software they can still improve performance by reducing the number of times the network stack needs to be traversed. Instead of multiple small packets traversing the stack individually, a large packet is passed through the stack and is segmented into smaller packets in one of the later stages of the network stack. [Foue].

2.2.2 Hardware interrupts

A NIC notifies the CPU when new packets have arrived through a hardware interrupt. Hardware interrupts can be important to take into account when trying to improve performance as they literally interrupt (preempt) the code (thread) that is currently running on the CPU, having a lot of interrupts can slow the entire system down, and create latencies [Hatb]. Therefore the processing done in hardware interrupt handlers is generally limited to the bare minimum and anything deferrable is typically deferred using a software interrupt [BC05] which we will see in Section 2.3.

Before the introduction of NAPI (formerly called New API) a NIC would create a hardware interrupt for every packet that arrived, but NAPI reduces the amount of hardware interrupts required by forcing the CPU into poll mode. When a new hardware interrupt is triggered, the CPU will temporarily poll to check if there are new packets with hardware interrupts of new packet arrivals disabled. We will see how this works in a bit more detail in Section 2.4.3.

The amount of hardware interrupts generated by a device per CPU are logged in the /proc/interrupts file which can look like this when printed, we only show the line of the NIC which in our case is named enp0s25:

```
$ cat /proc/interrupts
           CPU0       CPU1 CPU2 CPU3
 46:   18272  0    0    0    PCI-MSI-0000:00:19.0  0-edge  enp0s25
```

In this case the NIC only interrupts CPU0 and has done so 18272 times since boot.

2.3 Linux kernel basics

In this section we describe several structs, representations and mechanisms provided by the kernel, specifically those that we will later refer to.

Structs

In the rest of the text we will refer to some C structs defined in the kernel. Our goal here is to explain them on a high level so you can understand the rest of the text without requiring a deep understanding of network stack-related data structures.

- `sk_buff`: often abbreviated skb in a name, the "main" structure containing packet data in the network stack. It is allocated when a packet arrives on the NIC or when the user space application wants to transmit a packet
- `sockaddr`: a generic socket address that is not bound to a single address family
- `sockaddr_in`: a struct that fits in the same memory as `sockaddr`, the "in" stands for AF_INET: the Internet Protocol v4 address family. This struct is used for IPv4 sockets to define the source and destination, but as the socket related system calls are address family agnostic it is always cast to a `sockaddr` pointer before being passed
- `msghdr`: structure containing both a sockaddr (of the destination) and pointer(s) to the data of the packet
- `ethhdr`, `iphdr` and `udphdr`: protocol headers, they are defined in such a way that they are not padded/aligned and a pointer to the right location in packet data can be cast to them to make it easier to work with the headers

Software interrupts

A software interrupt is similar to a hardware interrupt, it is used to signal a CPU that it needs to complete some task, for example, process a packet that has arrived. But instead of a physical signal being sent to the CPU which interrupts it, software interrupts are handled

in software and are only checked at specific moments in time. Whenever a system call, which we will discuss next, is about to return to user space, or a hardware interrupt handler exits, any software interrupts which are marked pending (usually by hardware interrupts) are run [Foug].

Although being less invasive than hardware interrupts they still have an impact on performance as they defer processing the user space application to process the software interrupt.

The amount of software interrupt requests (softirqs) generated for every CPU process is logged in /proc/softirqs which can look as follows when printed, we only show the network-related NET_TX and NET_RX softirqs:

```
$cat /proc/softirqs
                CPU0       CPU1       CPU2       CPU3
NET_TX:          132        475        170        135
NET_RX:        19979    1671223         69        109
```

System calls, user space, kernel space and context switches

System calls are interfaces that allow user level applications to request services and functionality from the kernel [2]. These services can include actions like reading or writing files, creating new processes, managing memory, and interacting with hardware devices or more specifically for our use case: transmitting and receiving data over the socket interface which provides access to the Linux kernel network stack. System calls provide a controlled way for user level programs to access privileged operations and interact with the underlying hardware and resources of the computer. For our work the relevant system calls are: *socket()*, *bind()*, *sendto()*, *sendmsg()* and *recvmsg()*.

Virtual memory is segregated into two distinct areas: user space and kernel space, as the name implies the user application run in user space and the kernel runs in kernel space. When the user executes a system call we therefore have to switch from user space to kernel space temporarily and after the system call has been completed we need to change back to user space. These switches are called context switches.

These context switches can have a significant impact on performance as they cost at least several microseconds [LDS07]. This performance impact is caused by the CPU having to save the current state of the CPU, load the state of a new process and then resume executing the new process [Ash]. While the context switch itself has a cost, the further execution speed of the process(es) is also impacted as the CPU caches are (partially) filled with data from the old thread where we switched from. Context switches are not only caused by system calls but also in multitasking when the scheduler preempts a process to provide another process its share of the CPU time or when a hardware interrupt is received [Gar+].

To illustrate why several microseconds, while sounding small, has huge implications for network performance we calculate the time we have per Maximum Transmission Unit (MTU) sized packet. We assume that we want to saturate a 10 gbps link and that we have the standard MTU of 1500 bits.

$$10 \text{ gbps } / 1500 \text{ bits} = 666667 \text{ packets per second (pps)}$$

$$1 \text{ second } / 666667 \text{ packets} = 0.0000015 \text{ seconds} = 1.5 \text{ microseconds}$$

So to achieve line rate with this MTU we have to transmit one MTU-sized packet every 1.5 microseconds meaning that the cost of a single system call alone would already be too high.

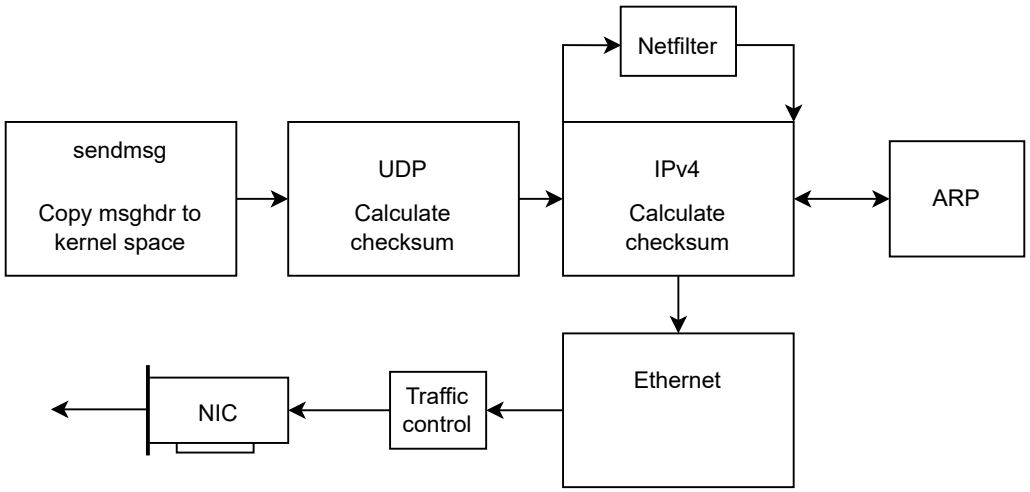

Figure 2.2: High-level flow when doing a *sendmsg()* system call

2.4 The basic UDP flow through the kernel

In this section we delve into the Linux kernel BSD socket implementation for UDP over IPv4. As we will be implementing our own alternative network stack that will use very similar function calls to create, bind, send, and receive messages we also take a quick look at the parameters of these functions. For the data flow through the kernel we leave out the complexity of discussing how everything is implemented exactly and instead focus on a high-level view while diving deeper into the parts that are of particular interest due to their performance impact or the functionality provided. The performance aspects we will look deeper into are mainly the ones discussed in Section 2.3 plus memory copies, hardware offloads and optimizations that have already been implemented in the stack.

2.4.1 Creating and binding a socket

A UDP socket can be created with a call to *socket()* the full call is the following:

```
socket(AF_INET, SOCK_DGRAM, 0);
```

We already filled in the specific arguments to create an IPv4 UDP socket: AF_INET stands for the Internet Protocol (v4) address family and SOCK_DGRAM basically tells the kernel that we want UDP as transport layer protocol.

and the bind() call is the following:

```
bind(int sockfd, (struct sockaddr*)&server_addr, sizeof(server_addr))
```

in this call the sockfd is the file descriptor of the socket we are binding, this is returned by the *socket()* call, the `server_addr` is a sockaddr_in with both the IPv4 address and port number we want to receive data on. And the last argument is simply the size of the passed address. Binding a socket associates it with a port number and one or all IP addresses. Any incoming packets on this port and, if specified, IP address will be directed to this socket.

2.4.2 Sendmsg()

The *sendmsg()* system call is the following:

```
sendmsg(int sockfd, const struct msghdr *msg, int flags);
```

The notable arguments are the sockfd which is the file descriptor of the socket and the `msghdr` which should be filled with the `sockaddr` of the destination and pointer(s) to the packet

data.

The overview of the flow inside the kernel is shown in Figure 2.2. When the user application makes a call to *sendmsg()* a context switch is done into kernel space and the packet data and destination in `msgshdr` are copied from user space to kernel space. For the rest of the explanation, we assume no type of segmentation offloading is used. The significant tasks in the context of UDP involve creating the header, which encompasses the computation of the checksum. If employing software-based checksum calculation the complete UDP checksum is determined at this point. Alternatively, if hardware offloading is enabled, the checksum will later be completed by the NIC. However, even if hardware checksum offloading is used, a part of the checksum, namely the checksum over the pseudo-header is always computed in software. Therefore it is important that the checksumming operation is implemented efficiently which is why the code responsible for software-based checksum calculation is tailored to the specific CPU architecture in use. For instance, on AMD64 (x86-64), an assembly code segment is employed to compute the checksum.

Once the UDP header has been filled we can pass the packet to the next layer, namely the IP layer. Here we build the IPv4 header and also compute its checksum which is always done in software [Foui]. Here we also make our first call to a Netfilter hook. We will now dive a bit deeper into Netfilter as this is a useful feature provided by the kernel that as we will show here is non-trivial to implement in a user space network stack due to its tight integration with the Linux kernel network stack. We also show on a high level how it works in order to be able to provide an alternative with our implementation.

Netfilter

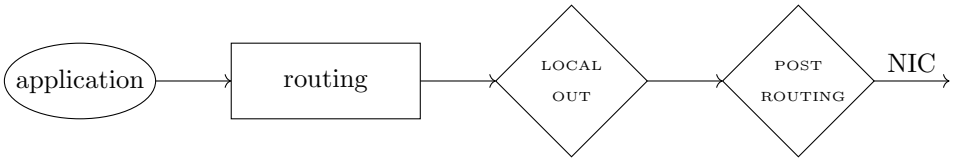

Figure 2.3: Netfilter hook locations in diamonds in the path from application to the NIC

Netfilter is a framework that provides packet filtering, network address translation (NAT), and other network-related functionalities in the Linux operating system. Netfilter is primarily used for firewalling and network traffic manipulation purposes such as Network Address Translation (NAT). Netfilter can be used through iptables or nftables[Net].

Netfilter has hooks that are called at different points in the Linux kernel network stack processing, for the output path from a user space application to a NIC the hooks are shown in Figure 2.3. The hook point LOCAL_OUT is executed for every packet coming from a user space application, while the POST_ROUTING hook is also run for packets being forwarded from another interface. When Netfilter is hooked several parameters are passed including the current packet (`skb` struct), socket context, and network device. The tight integration with the kernel network stack data structures and the specific hook points make it non-trivial to integrate this into a user space network stack.

Instead of diving into the detailed functionality of Netfilter and how it could potentially be modified to make it work in a kernel-bypass scenario we will learn about alternatives in Section 3.5.1 and our implementation in Section 4.2.6. We do this because these alternatives provide improved performance and do not require large modifications to kernel code to work together with AF_XDP.

When continuing in the path down to the NIC we are still in the IPv4 section in which the Address Resolution Protocol (ARP) that is also implemented in the kernel is used.

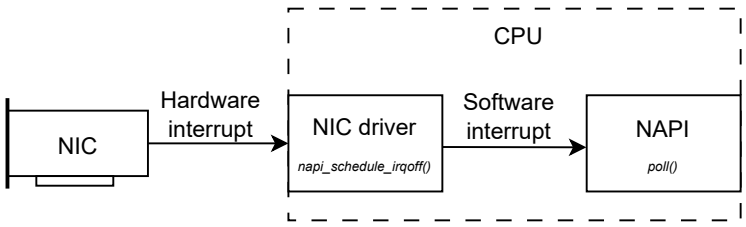

Figure 2.4: Hardware interrupt being generated on an incoming packet, which then raises a software interrupt inside the NIC driver to start the NAPI poll loop

Address Resolution Protocol (ARP)

The kernel network stack implements the ARP which maps IPv4 (and IPv6) addresses to MAC addresses in the local network. This is because in the local area network (LAN) hosts are identified by their MAC address while on the highest level IP addresses are used, or domain names which are mapped to IP addresses. It is not strictly necessary to implement this protocol to create a functional network as the user can manually generate a table with these mappings and keeps it up to date if hosts enter or leave the network. In reality, this might not always be feasible due to the large number of hosts in a network and frequent changes. If we create our own network stack we would therefore have to implement the ARP or, as we will see in our implementation section, figure out a way to utilize the kernel-provided ARP implementation.

ARP uses simple messages on top Ethernet to communicate, we can check the Ethertype inside the Ethernet header to see if a packet is an ARP message.

The last protocol layer our packet passes through is the Ethernet layer, other than filling the header nothing of special interest happens. But the Ethernet layer does not pass the packet directly to the NIC, instead it passes through traffic control first.

Traffic control

The primary goal of traffic control in Linux is to manage the bandwidth and latency of network connections, so that critical applications or services receive the necessary resources while preventing any single application from overwhelming the network and causing degradation in performance for other applications.

This is done through queueing discipline (qdiscs) which are basically an algorithm to decide in which order data is queued to be sent on the NIC. Fq_codel, which is currently the default qdisc does this intelligently by using a stochastic model to classify incoming packets into different flows. It provides a fair share of the bandwidth to all the flows using the queue. The usage of traffic control and qdiscs can have a significant effect on the performance of network data transfers when multiple flows are active [NW13].

Once the packet is queued, the NIC will eventually read the packet data from main memory using DMA and transmit it. And this concludes our complete packet transmission. We will now take a look at the reverse scenario where a packet is received.

2.4.3 Incoming packet(s) on the NIC

Unlike packet transmissions, packet receptions do not start with a system call by the receiver because packets can arrive at any moment. When a packet arrives on the NIC, the NIC places the packet in a RX ring in main memory using DMA and notifies the CPU by sending a hardware interrupt. This hardware interrupt is handled in the NIC driver by raising a software interrupt, which, when the software interrupt is being handled calls a NAPI poll loop to temporarily poll the RX queue while disabling further hardware interrupts. A schematic overview can be seen

in Figure 2.4. The advantage of this NAPI poll method is of course that both the number of hardware and software interrupts are minimized.

Inside the NAPI poll loop packets are also pushed through the network stack. The traversal happens in the opposite direction of the one shown for *sendmsg* and passes through the Ethernet, IPv4 and UDP implementations while also calling two Netfilter hooks. The last step here is the received packet being placed in the socket receive buffer. The user space application can then call *recvmsg()* to receive this data.

2.4.4 Recvmsg()

The recvmsg call is the following:

```
recvmsg(int sockfd, struct msghdr *msg, int flags);
```

With the notable arguments being the socket file descriptor and a pointer to a `msghdr` struct which will be filled with the packet data and address of the sender.

As we have seen in the previous section how the data enters the socket buffer through the network stack we already know that the *recvmsg()* call, unlike the *sendmsg()* call does not traverse the network stack. Instead what *recvmsg()* does is simply copy the data from the socket receive buffer to the region in user space. Therefore the only memory copy in the receive path happens here.

2.4.5 Performance considerations

We finish our discussion of the Linux kernel network stack by discussing some details of where packets are processed on multi-core systems and we also discuss an optimization already implemented in the kernel to reduce the amount of system calls.

Multi-core systems

For multi-core systems we ask ourselves an important question: “On which CPU are the packets processed on a multi-core system?”. For transmitting packets the processing of packets happens on the CPU running the user space application. But what about incoming packets? This is important as we have seen that receiving packets causes both hardware and software interrupts, and causes the CPU core to use NAPI polling to retrieve extra packets temporarily. This makes it so our user space application is temporarily interrupted if it is running on the same core.

In the situation that the NIC only has a single receive queue, incoming hardware interrupts are handled by a single CPU and this is also the CPU that handles the software interrupt and the NAPI poll loop. Additionally, when using the kernel network stack, this CPU also pushes the packets through the kernel network stack and into the receive buffer of the socket. While modern NICs support multiple queues and can divide the load over multiple CPUs [Hatc], the CPU handling the interrupts and the one running the user space application can still differ. Therefore it is not a bad idea to also take the (CPU utilization) of the interrupt handling CPU into account when considering network stack performance.

sendmmsg() and recvmmsg()

The Linux kernel network stack already includes some optimizations. One of them are *sendmmsg()* and *recvmmsg()* which are alternatives for *sendmsg()* and *recvmsg()* respectively. They can reduce the amount of system calls that need to be made to transmit packets by passing multiple packets in a single system call. As we have seen in Section 2.3: the context switches caused by system calls can have a significant performance impact. *sendmmsg()* and *recvmmsg()* work by utilizing a struct `mmsghdr` instead of the standard struct `msghdr` which serves the same purpose but can contain multiple packets intead of just one.

Behind the scenes these functions make multiple calls to *sendmsg()* and *recvmsg()*, which copy the data one `msghdr` at a time between user space and kernel space or vice versa. So other than a reduction in system calls these function do not provide further optimizations.

2.5 eBPF eXpress Data Path (XDP)

eXpress Data Path (XDP) is a high-performance datapath utilizing eBPF. It's an eBPF data path used to transmit and receive packets by bypassing the operating system network stack. Along with XDP a new address family, AF_XDP, was introduced, it allows an eBPF program with XDP to forward packets to an AF_XDP socket, effectively bypassing the network stack and passing data directly to user space. Using this type of socket the user space program can also transmit data while bypassing the kernel network stack.

In this section we take a look at what eBPF is, how XDP and AF_XDP works and the necessary details to understand the performance implications of some of its options such as the attach mode and zero-copy mechanism. As well as a high-level understanding of how packets can be transmitted and received over an AF_XDP socket in order to optimally use this in our implementation.

2.5.1 eBPF

eBPF is a feature in the Linux kernel that allows for programmable data path processing and packet filtering. Originally derived from the traditional Berkeley Packet Filter (BPF), eBPF extends its capabilities to provide a more flexible and efficient mechanism for dynamic instrumentation, observability, and tracing within the kernel. It works by attaching an eBPF program to a designated code path in the kernel, when the code path is traversed, any attached eBPF programs are executed.

eBPF programs are small, sandboxed bytecode programs that are safe to execute within the kernel. These bytecode programs can be loaded into the kernel via system calls or user space utilities. You can write eBPF code in a pseudo-C style which can then be compiled to bytecode (or you could write bytecode directly). Then this bytecode can be loaded into the Linux kernel using the bpf system call. As the program is loaded in the kernel it is first verified to be safe to run this means:

- The program loading the eBPF program has the correct privileges to do so
- The program does not crash or otherwise harm the system
- The program always runs to completion (it cannot enter an infinite loop)

These restrictions might limit the capabilities of eBPF somewhat, the verification step makes it so eBPF is not Turing-complete [Bau]. But these restrictions are essential to allow user space applications to load code into the kernel, if we could load unsafe code into the kernel we could cause the entire system to crash or cause the kernel to go into an infinite loop effectively freezing the entire system. But due to the Just-in-Time (JIT) compilation to translate the generic bytecode into machine-specific instructions, eBPF programs run as efficiently as natively compiled kernel code or as code loaded as a kernel module [eBP; Fle].

2.5.2 XDP

XDP's name comes from the fact that it gets access to the packet data early on in the processing pipeline. Where exactly this happens depends on the mode that is being used. There is a general mode called SKB which does not require driver support, therefore, making XDP available without any modifications to the NIC or driver. If however support is added to the NIC driver we call this NATIVE or driver mode and this has an advantage. As can be seen in Figure 2.5 the point at which the eBPF XDP program gets called differs, for NATIVE mode

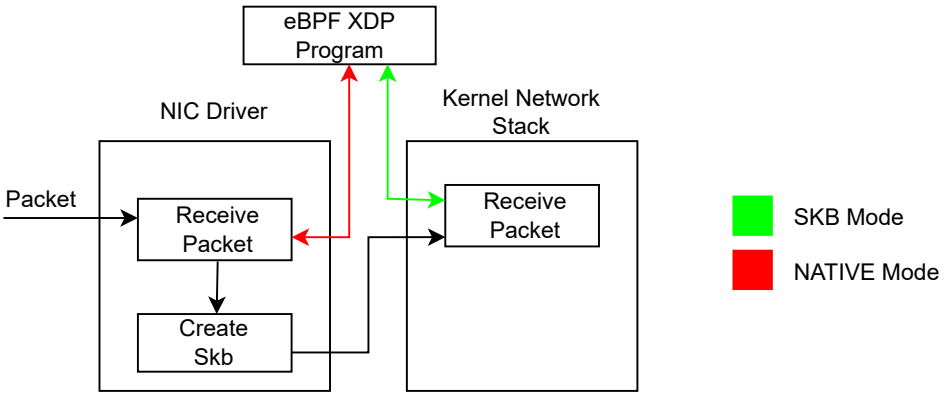

Figure 2.5: Code flow locations where the eBPF XDP program is called for SKB and NATIVE mode

this is before an `sk_buff` has been allocated while for SKB mode this is after an `sk_buff` has been constructed and passed to the Linux kernel network stack. The green and red arrows are two-directional as the XDP program returns which action to perform on the packet.

In terms of packet handling within a XDP eBPF program, the frame is passed to the program and can then be read and/or modified and a value is returned to indicate what needs to happen to the frame. There are several options available [Cil]:

- XDP_ABORTED: indicates an error condition, behaves like XDP_DROP but also passes an trace_xdp_exception tracepoint which can be probed to detect (and debug) misbehavior [Des]
- XDP_DROP: drop the packet right away
- XDP_PASS: pass the packet through the Linux kernel network stack (like normal)
- XDP_TX: transmit the packet out of the same NIC it just arrived on
- XDP_REDIRECT: similar to XDP_TX but allows you to transmit the XDP packet through another NIC, this option can also be used to redirect into a BPF cpumap meaning that the packet can be pushed to another CPU (core) for further processing, this can also be used to redirect the packet to an AF_XDP socket

2.5.3 XDP address family (AF_XDP)

The XDP address family (AF_XDP) allows the redirection of data frames to user space applications using an AF_XDP socket (xsk). Every xsk has two associated rings: an RX (receive) ring and a TX (transmit) ring and every xsk is associated with exactly one UMEM, but a single UMEM can be associated with multiple xsk. A UMEM is a region of virtual contiguous memory, divided into equal-sized frames. There are two rings associated with the UMEM: a FILL ring and a COMPLETION ring. When creating the UMEM you need to define the sizes of the frames, and how many frames there are. You can use a custom value or the default (currently 4096) but you need to accommodate the largest possible packet that could be sent or received. While also keeping in mind that by default the UMEM operates in aligned frame mode meaning that the addresses will always be mapped to multiples of the frame size, for example with the default frame size 4096, addresses will always be aligned to multiples of 4096.

A later patch allows unaligned mode, which provides some advantages such as making it possible to place frames inside the UMEM wherever you please but also complicates things while also having a small impact on performance (up to about 1,5%) [Laa].

The functions of the rings are the following:

- COMPLETION: kernel space fills this rings with the UMEM frames that have been transmitted
- FILL: user space fills this ring with UMEM frames that can be used by kernel space to write newly received packets into
- RX: kernel space fills this ring with UMEM locations plus the length of received packets
- TX: user space fills this ring with UMEM locations plus the length of packets it wants to transmit

A UMEM can be visually represented as can be seen in Figure 2.6. The large rectangle represents the entire block of memory which is the frame size times the amount of frames.

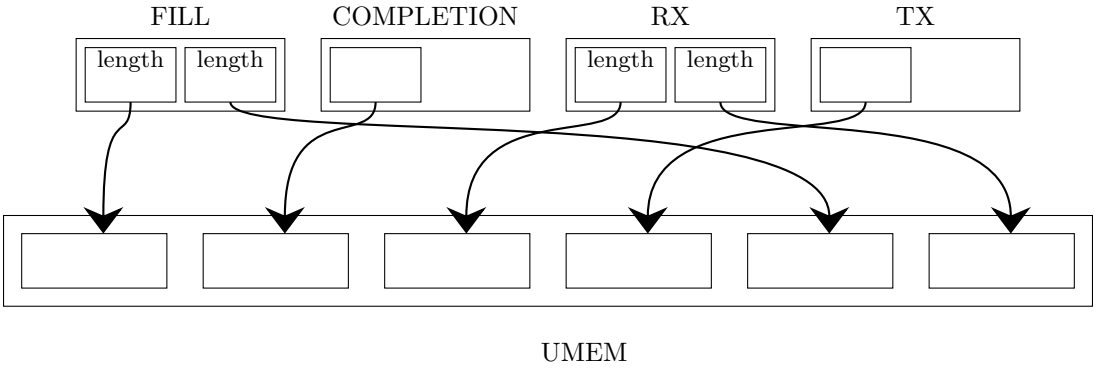

Figure 2.6: Visual representation of UMEM plus rings with the arrows representing the pointers to the UMEM addresses

	user space	kernel space	contains
COMPLETION	consumes	produces	address
FILL	produces	consumes	address
RX	consumes	produces	address and length
TX	produces	consumes	address and length

Table 2.1: All the AF_XDP rings, who consumes and produces and what they contain

The rings are summarized in Table 2.1. The RX, TX, FILL and COMPLETION rings can be divided into two types as seen from the perspective of user space: consumer ring and producer ring where user space consumes and produces respectively. Depending on the ring the items contain just a pointer to a UMEM location or also contain the length of the packet.

In summary all the rings (assuming a single AF_XDP socket) and UMEM can look like Figure 2.6, where our rings have length 2 (frames) and our UMEM has length 6 (frames). In reality you would like to avoid a situation such as the one shown. If in the shown situation a new packet arrives, the NIC driver in kernel space can take a packet from the FILL ring to fill, but it cannot put another packet in the RX ring, therefore no new packets can be received. You need to make sure your rings are large enough and that your user space application process clears the RX and COMPLETION ring fast enough as well as filling the FILL ring fast enough.

A general recommendation for the FILL ring size is the size of the RX ring plus the size of the hardware RX ring size in the NIC. Assuming you also add new frames to the FILL ring fast enough this ensures that the NIC driver will never run out of memory (locations) to put new incoming packets into. This because NIC drivers have not been designed with memory allocation issues in mind, and if they happen they can be quite expensive. It was designed this way as while using the kernel network stack, kernel allocated memory is used that only runs out in Out Of Memory (OOM) situations which should be rare [xdp].

Transmitting a single packet

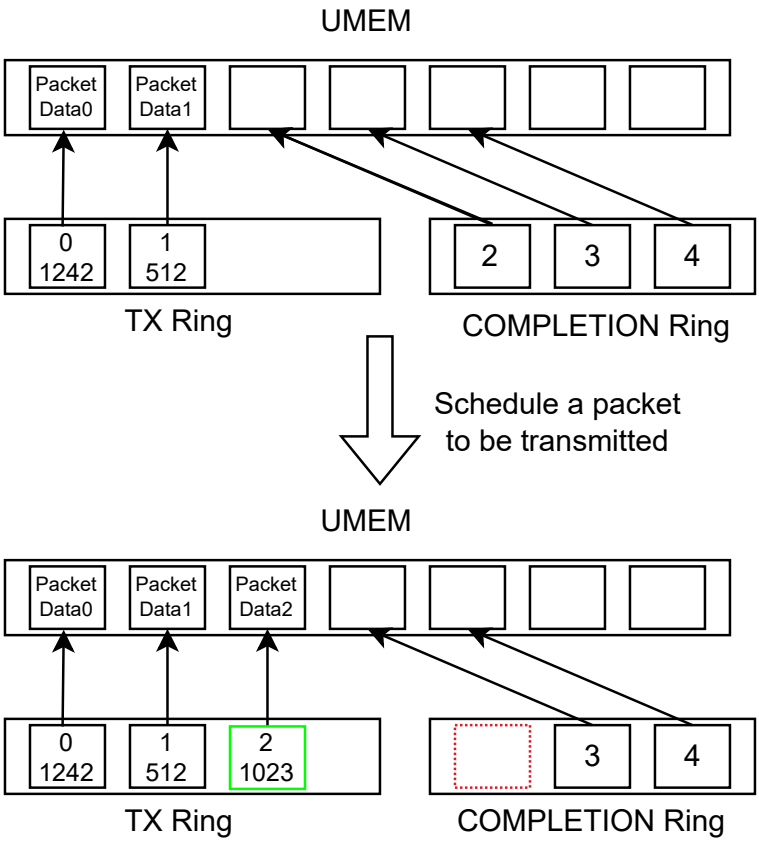

Figure 2.7: Simplified view of a UMEM, TX ring and COMPLETION ring before and after scheduling a packet for transmission on an xsk

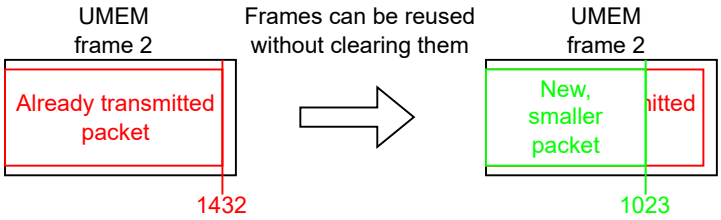

Figure 2.8: UMEM frames can be reused without clearing or zeroing out the old data

We visualize the process to transmit a packet using an AF_XDP socket in Figure 2.7. We must always obtain a UMEM frame address to fill with our packet, to grab one we can consume one from the COMPLETION ring, which contains frames that contain(ed) packets that have been transmitted by the NIC and are ready to be reused. In the figure the COMPLETION ring provides us with UMEM frame 2. We then fill the UMEM frame with a packet (including headers) and submit its location as well as its size to the TX ring. After we have done this all we must wake up the NIC driver for it to actually transmit the frame. Waking up the NIC driver is done through a *sendmsg()* system call (*poll()* can also be used but is slower [3]). Unlike when using the kernel network stack we don't pass any packet or destination with the *sendmsg()* call: this call does not result in any data copies or any packets being processed by the kernel network stack. The NIC driver will use the location to grab the packet data and

because we passed the length of the packet, it also knows where the packet stops. This makes it so we do not have to zero out the UMEM frame as it might be larger than the packet and still contain part of an old packet due to being reused as is illustrated in Figure 2.8.

When the NIC (driver) transmits the packet it reads its location in the UMEM from the TX ring and once it is done transmitting places a reference to the UMEM frame in the COMPLETION ring so it can be reused again by the user space application.

Receiving a single packet

Now that we know how we can send frames, we also take a look at receiving frames of which the process is visualized in Figure 2.9. To check if there are new frame references available we can peek the RX ring and if there is one available grab it. If there aren't any available we have to wake up the NIC driver in order for it to update the RX and FILL ring, we do so by utilizing a *recvfrom()* system call, and just like the *senmdsg()* system call we use when transmitting: we do not pass any buffers so no copies or real network stack processing happens because of it. We then grab the UMEM location and packet length (which the NIC filled in) from this frame reference and utilize it to process the packet, this processing should include basic header checks: is the frame destined for us? Are the checksums correct? Please note that you could also choose to implement some or all of these checks in the eBPF program that forwards to packets to our AF_XDP socket. The advantage is that when using NATIVE mode you could drop the packet directly in the NIC driver before any real processing has been done.

Next, if everything is correct we can pass the packet data to the application running on top of our xsk, which in our case will be QUIC. For AF_XDP we are free to do with this UMEM frame as we please. After we are done with the UMEM frame we submit a reference to it to the FILL ring so the frame can be reused by the NIC driver to put another incoming packet into.

Zero-copy

When utilizing the NATIVE attach mode we can additionally enable zero-copy if the NIC driver supports this. What this does is make the NIC driver aware of the location of the UMEM and its frames so when receiving packets the NIC can DMA packets directly into the correct UMEM frame instead of the packet first having to put in a buffer from the RX ring of the network device in main memory and then having to be copied to the UMEM. The difference is visualized in Figure 2.10 for the receive flow. The transmit process is virtually the same but in the opposite direction, when utilizing zero-copy mode the NIC can directly pull the packet from the UMEM frame, while without it the packet must be copied from the UMEM frame to a network device TX ring location in main memory.

Waking up the NIC driver

When describing how to send and receive packets using AF_XDP we included waking up the NIC driver in the process and as we explained this is done through a system call. However there is an optimization possible. Sometimes the NIC driver might already be running, for example when in the NAPI poll loop. To prevent us from having to wake up the NIC if this is the case there is a flag that can be set when creating an AF_XDP socket, namely XDP_USE_NEED_WAKEUP. When this flag is set the user space application can use a libxdp provided function to check if it needs to wake up the driver before doing so. If it does not, it saves the user space application from performing a relatively costly system call.

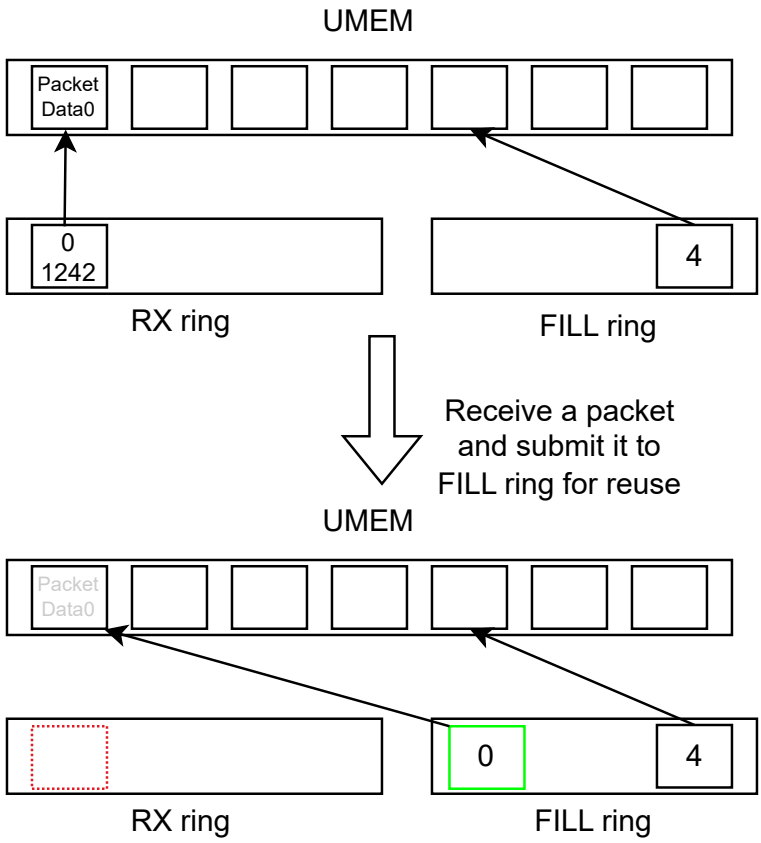

Figure 2.9: Simplified view of a UMEM, RX ring and FILL ring before and after receiving a packet an xsk

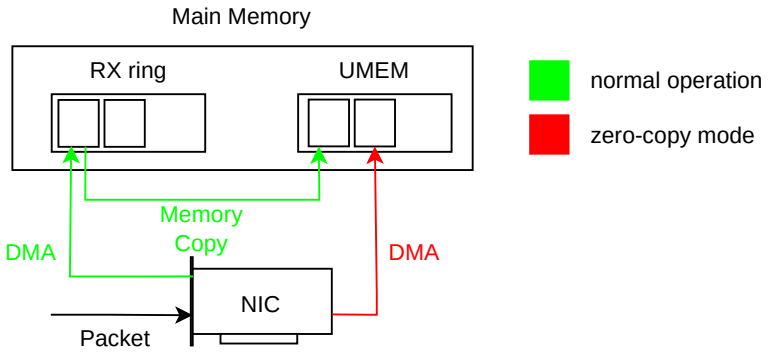

Figure 2.10: Normal operation vs zero-copy mode when receiving a packet

Chapter 3

Related work

In this chapter we look at multiple forms of related work. We look at several publications which are related to host network stack and QUIC performance, including research into hardware techniques like Direct Cache Access (DCA). Further, we also look at XDP-related publications, XDP for Microsoft Windows and several alternatives that provide functionality such as kernel-bypass similar to that of AF_XDP.

3.1 Host network stack optimizations

Cai et al. [Cai+21] discuss host network stack overheads, specifically for the Linux kernel using 100 gbps links. They mention that while NIC link throughput speeds have increased by 4 - 10x over the past few years, all other host resources including CPU speeds, cache sizes and NIC buffer sizes have largely been stagnant. As a result the need has emerged to create a more (CPU) efficient host network stack. They do a deep analysis of the effects of resource sharing (due to CPU cores in the same NUMA node), the impact of in-network congestion, flow sizes, DCA, IOMMU and congestion control protocols. The main future directions they see as a result of their research, which is mainly based on iperf runs over TCP, are zero-copy mechanisms and CPU-efficient transport protocol design.

3.2 Direct Cache Access (DCA)

DCA is a technique similar to DMA but instead of the NIC being able to write data directly to the system's main memory the NIC can write data directly to the CPU cache. Intel's implementation of this is called Data Direct I/O (DDIO), works for all I/O devices and is invisible to software meaning that driver changes are also not required [Intc]. Wang et al. [WXW22] develop an analytical framework to predict the effectiveness of DCA under certain hardware specifications, system configurations, and application properties. They conclude that while DCA can have performance benefits, Intel's implementation also has disadvantages such as it being hard to balance low memory traffic, low cache usage and resistance to fluctuations in workload. As for low memory traffic and low cache usage the buffers need to be relatively small, but this causes the buffer to fill completely if the workload fluctuates.

3.3 Alternative network stacks and kernel bypass techniques

Chen and Sun conducted a survey about kernel bypass [CS18], they give a (very) high level overview of the path a packet takes when it arrives on the NIC to the user space application.

They classify the kernel network stack overheads into three categories:

- System call overhead, the cost of switching from user space to kernel space and back which may cause cache pollution, Translation Lookaside Buffer (TLB) flush.
- Extra data copying, namely the copy or copies between the user space and kernel space buffers
- Per-packet processing: the general processing (routing, netfilter, checksum calculation, ...) that happens for every packet.

They describe and compare several alternatives to the Linux kernel network stack which include DPDK, mTCP and Netmap.

3.3.1 Data Plane Development Kit (DPDK)

Data Plane Development Kit (DPDK) [Foua], is a set of libraries and drivers that enable high-performance packet processing in networking applications. It is similar to AF_XDP in the sense that it allows to bypass the kernel network stack and allows an implementation of a custom network stack in user space.

DPDK applications use a polling model, where they actively poll network interfaces for incoming packets instead of relying on interrupt-driven approaches used by traditional networking stacks. This reduces interrupt handling overhead but does cause the CPU core to be constantly busy with polling, meaning that CPU consumption is always 100% even if no packets are being processed.

Later on an AF_XDP based poll mode driver was also added to DPDK which basically runs DPDK on top of AF_XDP with busy poll mode. This is a disadvantage over using AF_XDP directly as this also allows to use of interrupts and NAPI logic to only poll a short while after an interrupt has been received which does not cause our CPU utilization to stay at 100% all the time [Inta].

3.3.2 Netmap

Netmap by Rizzo [Riz12] is a framework for very fast packet I/O from user space. In building Netmap, they identified and reduced or removed three main packet processing costs:

- per-packet dynamic memory allocations
- system call overheads
- memory copies, eliminated by sharing buffers and metadata between kernel and user space

This is achieved as follows: they remove the memory copies by cutting off the standard kernel network stack from the NIC and mapping the NIC memory buffers directly to user space. As these buffers are reused over and over again they are only allocated once, avoiding per-packet memory allocation. Netmap also reduces the number of system calls by allowing the transmission of multiple packets per system call.

Netmap works similarly to how AF_XDP sockets work: the user has access to a large buffer in which frames can be placed, inside a Netmap ring the packet sizes are placed and to signal the NIC to transmit the frames a system call is made. However a disadvantage of their approach of cutting off the NIC is that while the kernel network stack can still be used, hardware offloads can no longer be used by the kernel network stack [Gim].

3.3.3 mTCP

mTCP [Jeo+14] is a scalable user space TCP stack. In the architecture of mTCP, a distinct TCP thread undertakes the responsibility of protocol processing. While the mTCP application operates on a user-level socket, engaging communication with the TCP thread via a shared

buffer. Consequently, resource-intensive system calls are eradicated. Instead, mTCP employs lightweight user-level function calls to fulfill the demands of TCP/IP processing. It extends the PacketShader I/O engine [Han+10] in order to bypass the kernel network stack.

Additionally, mTCP employs batching techniques to offset the impact of context switches between application threads and TCP threads. Beyond system calls, batching is also implemented at the packet I/O level to diminish the overhead associated with processing each packet.

The process of migrating an existing application to mTCP necessitates adjustments from the standard socket API to the mTCP socket API which has functions that are similar in name and function making it easy to migrate an application to mTCP (for example *accept()* and *mtcp_accept()*). According to their measurements mTCP has the potential to accelerate the performance of existing applications by up to a factor of 3.

3.3.4 Onload

Onload is a commercially developed product offering a high-performance user-level network stack, which accelerates TCP and UDP network I/O for applications using the BSD sockets on Linux [Xil]. The original way this was intended to work is with a specific hardware interface called ef_vi which is provided by select Xilinx network adapters (NICs). This zero-copy interface provides user space with direct access to the data structures of the NIC. With OpenOnload a user-level shared library is included that intercepts network-related system calls and automatically converts them to calls to the Onload network stack. Utilizing this library you can change the network stack from the kernel network stack to the Onload network stack without changing any networking code.

Compatibility with AF_XDP is at the time of writing under development and is not yet at final release quality. But because of this, it is possible to use OpenOnload on non-Xilinx NICs.

3.4 QUIC

Jaeger et al. [Jae+23] introduce a benchmarking framework for QUIC on top of QUIC interrop runner [See] which they use to compare the performance between different QUIC implementations as well as the impact of different hardware offloads, buffer sizes and AES NI acceleration.

They draw several conclusions. The first is that performance of (interoperating) QUIC libraries differs significantly: the best performer, lsquic both in the server and client achieves 3000 Mbps while the slowest combination which is picoquic as server and mvfst as client only achieves 52 Mbps in their test setup. They inspect CPU cycles spent on each task and conclude that packet I/O is the most expensive task taking up about 65% of the CPU cycles for lsquic. Their third conclusion is that the default UDP receive buffer sizes are too small and increasing their size improves performance. Their next conclusion is that while crypto is also a considerable cost for QUIC, using a faster cipher does not impact performance significantly while it does improve performance significantly for TCP/TLS indicating that packet I/O forms a bottleneck for QUIC. Further they conclude that none of the QUIC implementations tested benefit from segmentation offloads and that they cannot get sendmmsg or recvmmsg working under lsquic. Their final conclusion is that the used hardware is highly relevant because the performance differs significantly between different processors meaning that to compare performance you need to use the same hardware.

3.4.1 In-kernel QUIC

Wang et al. develop and test an in-kernel implementation of QUIC [Wan+18]. They state that they expect that the achievable performance with a user space implementation might be held back because every message triggers a context switch. They use both a real-world wireless

connection as well as a virtual connection with simulated packet loss to test their implementation against the TCP implementation in the kernel. They conclude that in most scenarios QUIC is faster than TCP.

3.4.2 QUIC on top of DPDK

Tyunyayev et al. implement picoquic, a QUIC implementation, on top of DPDK [Tyu+22]. They identify three inefficiencies in the kernel network stack: interrupts, the utilization of the sk_buff structure, and the memory copy between user space and kernel space. They implement a simple UDP stack compatible with both IPv4 including ARP as well as IPv6. They measure a goodput that is more than three times higher than picoquic running in user space. Additionally, they measure that the relative CPU utilization spent on packet I/O is reduced from 18.7% to 5.7%. They also show that the achieved goodput performance is slightly higher than TCP.

3.5 XDP

Karlsson and Töpel [Kar18] discuss various improvements for AF_XDP with the goal to achieve speeds similar to DPDK. These suggestions made their way into (lib)XDP and the Linux kernel but not exactly as described. Their suggestions are: a built-in XDP program, XDP optimizations, Multiple TX rings for one UMEM, forcing in-order completion in order to cut out the COMPLETION ring, busy polling, and return trampoline (retpoline) optimizations. The retpoline mechanism in the compiler mitigates Spectre v2 type of attacks but also makes code less efficient, the performance degradation can be reduced by limiting the number of indirect function calls. In C these indirect function calls are mainly caused by using function pointers.

The busy poll options is now supported for AF_XPD, just like multiple TX rings (in the form of multiple xsks) have also been implemented in the kernel. And as they mention this also has the advantage that you can utilize the QoS and shaping support present in many NICs. Libxdp also provided a built-in XDP program and is more or less implemented as explained. Some retpoline optimizations have also been implemented [Lar].

3.5.1 XDP applications

Several applications of XDP have appeared or have been proposed. Miano et al. present [Mia+19] “Securing Linux with a Faster and Scalable IPtables” in which they use eBPF XPD to implement an alternative to IPTables (Netfilter). They preserve the IPTables filtering semantics making a true replacement that does not require rewriting IPTables rules. They benchmark their implementation and measure a performance that in general is better than IPTables.

3.5.2 Windows XDP

Windows also has an eXpress Data Path similar to that of Linux, during the writing of this thesis version 1.0.0 was released [Micc]. Just like on Linux it has two modes, a generic mode which intercepts the packets further down the line in the data path, and a native mode which requires driver support but intercepts the packets in the device driver. Although eBPF for Windows is currently not yet released there is a work-in-progress version available [Mica] so support for this can be expected in the future.

We took a look at the code and sample program [Micd] to see how similar it is to the Linux implementation and what its main differences are. We see that the Windows XDP API, although not sharing any underlying code with the Linux version, is similar to the Linux version. In their example program they have TX, RX, FILL and COMPLETION rings of type XSK_RING, a

UMEM with configurable frame size and total size. To create an XDP program the implementation uses the `XDP_RULE` type with specific filters. These are passed to the *XdpCreateProgram()* function, which constructs the program.

3.6 Other options in the Linux kernel

A seemingly similar option to AF_XDP sockets are AF_PACKET sockets, which also forward the packet to user space before passing through the network stack [1]. The one big difference is that the packet is also still forwarded through the network stack. This is useful for packet sniffers and analyzers, for example the packet capturing library: libpcap used by Wireshark [Gro]. AF_PACKET has gone through several revisions and additions such as the PACKET_MMAP facility that can provide great performance benefits [Fouk]. Another option in the Linux kernel are IPv4 raw sockets. These are similar to AF_PACKET sockets but lack layer 2 (link-layer) headers [Foub].

While we will use a kernel bypass method to achieve zero-copy support, the Linux kernel actually already supports the MSG_ZEROCOPY flag for TCP and UDP socket send calls [Fouj]. This also has the potential to improve performance just like zero-copy with AF_XDP [BD17]. However, it does have negative consequences due to the way it is implemented. It used page pinning to lock the user space memory buffer and allows the kernel space to access and utilize it as if it were in kernel space. Once the kernel is done with the memory it is released back to user space. This method however has a significant performance impact [Cor]. Making it only viable when memory buffers of 10 KB or larger are passed in send calls. This is in contrast with the AF_XDP zero-copy option which does not have an additional locking mechanism that causes an extra cost.

Chapter 4

Implementation

In this chapter, we discuss the implementation we have made. We start off by explaining our choice to use AF_XDP over other possible technologies that could allow us to achieve the same or similar results. Then we describe our implementation and discuss some performance-related things such as batched sending and the default buffer sizes. We also explain the API and compare it to the one of the UDP socket in the Linux kernel. We also explain an application utilizing the Linux kernel socket(s) can be converted to utilizing our implementation. This illustrates how our network stack can be used as an almost drop-in replacement.

4.1 The choice for AF_XDP

As we have seen in Chapter 2 the main goal of AF_XDP is to provide a kernel bypass path to directly forward packets as soon as they arrive on the NIC (driver) to a user space socket. The main reason for this is performance.

When discussing our related work in Chapter 3 we came across several alternatives. These include DPDK, Netmap, and the PacketShader I/O engine used by mTCP.

We chose AF_XDP over these due to its good integration with the Linux kernel and eBPF, for example it does not require custom drivers and it keeps the NAPI driver functionality. Due to AF_XDP being relatively novel compared to the other techniques it also has had less prior work done in terms of implementing network stacks directly on top of it. While for example DPDK already has several user space network stacks available including F-stack [Ten] and UDPDK [Lai+]. A disadvantage of AF_XDP is its performance compared to alternatives using only polling instead of interrupts.

4.2 AF_XDP UDP socket

For our custom UDP socket implementation we use AF_XDP and we implement the entire network stack, namely the three layers we use: Ethernet, IPv4 and UDP in user space. For this we make several design choices, but in general, we try to make our stack as simple as possible. We reuse kernel-provided structs and functions where possible. This has two advantages. The first is that it is easier to reuse already created structs and functions instead of having to redo this work ourselves. The second reason is that we minimize differences caused by a different implementation of for example checksumming.

We also keep the interface of our socket implementation relatively close to that of the kernel, in general the interface provided to the user is the same as with the kernel network stack except that function names are prepended with "AF", for example *socket()* becomes *AF_socket()*. We

will now compare our UDP implementation with the one provided by the kernel and note any differences.

4.2.1 Creating the socket

We provide the function *AF_socket()* for the user to create an AFXDPUDP socket as we call it, we return a struct `AFXDPUDP_socket` pointer instead of a file descriptor like the kernel does. The function definition is the following:

```
struct AFXDPUDP_socket* AF_create_socket(
                        struct AFXDPUDP_socketOptions * sockOpts)
```

Another difference compared to the kernel provided *socket()* function which we described in Section 2.4.1 is that we do not take the address family and transport layer type as arguments. This is because we only support IPv4 and UDP. Instead of also translating the *setsockopt()* system calls to our implementation we allow the user to pass a `socketOptions` struct. We do this as some of these options are necessary at the time the socket is set up and can not be changed afterward without having to destroy the underlying socket and UMEM followed by reconstructing them again.

For the user the exact values in the `AFXDPUDP_socket` pointer are not of importance, they are simply used to store state, a pointer to the UMEM, a pointer to the xsk etc.

In the *AF_bind()* call we initialize the socket further, the user passes the `AFXDPUDP_socket` pointer, the name of the interface and the (UDP) port to be used. The function definition is the following:

```
AF_bind(struct AFXDPUDP_socket* sock, const char * interface, u16 port)
```

Further the user provides the interface name of the network interface it wants to bind the xsk to and the (UDP) port number on which we will send and receive data. We also need to know our own IPv4 address and MAC address of the given interface when transmitting. But instead of letting the user provide them to utilize the kernel network stack to grab them. To do so we create a socket and utilize *ioctl()* with the file descriptor of the socket to extract these values as follows:

```
1 int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
2
3 struct ifreq ifr;
4 strncpy(ifr.ifr_name, interface_name, IFNAMSIZ - 1);
5
6 ioctl(sockfd, SIOCGIFHWADDR, &ifr);
7 // ifr.ifr_hwadr now contains the MAC address
8 ioctl(sockfd, SIOCGIFADDR, &ifr);
9 // ifr.ifr_addr now contains the IPv4 address
```

We store these values in the `AFXDPUDP_socket` struct so we don't have to make these system calls over and over again every time we need these values. This method does work for our setup but might not work in general as an interface can have multiple IP addresses and the user might want to use a specific one.

4.2.2 Sending packets: AF_sendmsg()

The function definition is the following:

```
AF_sendmsg(struct AFXDPUDP_socket* sock, struct msghdr* message);
```

The interface is identical to the one provided by the kernel except that we require a `AFXDPUDP_socket` pointer to be passed instead of a file descriptor and also don't support any flags.

The process used for transmitting packets is the one explained in Section 2.5.3. We construct the packet utilizing the `msghdr` struct from our function arguments which provides us with the

packet data and destination. To construct the outgoing packet we need to build three protocol headers: Ethernet, IPv4 and UDP.

We use the structs provided by the kernel for this, namely: `ethhdr`, `iphdr` and `udphdr`. The `ethhdr` is placed at the beginning of the UMEM frame, the `iphdr` directly after and the `udphdr` directly after the `iphdr`. After the UDP header we copy the data provided by the `msghdr`.

Next we describe the operations per header (protocol), which data we fill in and where we get it from.

Ethernet

In the Ethernet header we need to set several values: first the destination MAC is set. To get the destination MAC we look up the IPv4 address passed inside the `msghdr` in the ARP map that we store in our application as we will explain in section 2.4.2. The EtherType is always the same for us, namely the IPv4 EtherType. For the source MAC address, we use the one extracted during the *AF_bind()* call.

IPv4

To fill the values in the IPv4 header we use the IPv4 addresses provided inside the `msghdr`, for the source address we use the one extracted during the *AF_bind()* call. Further we also set the IP version to 4, the correct header length, the TTL to the default, the protocol to UDP, and all the other fields we haven't mentioned to 0 as we don't use them.

The checksum is calculated over the entire IPv4 header with the checksum field set to 0. For this we copy the functions from the Linux kernel this ensures three things:

- We can assume that the calculation is correct as it has been in use for many years.
- We ensure our checksum is compatible with that of IPv4 sockets running in the Linux kernel.
- We ensure that the computational complexity of the checksum algorithm is the same for our AF_XDP implementation and the kernel implementation. Therefore we can be certain that the chosen checksum algorithm is not the cause of any potential performance differences (assuming both the kernel and AF_XDP implementation calculate the checksum in software).

UDP

For the source port we use the value provided by the user or if the user has not provided one a default value. The destination port is provided in the `msghdr`. For the length, we add the header length to the length of the data provided inside the `msghdr`.

Just like for the IPv4 header we use checksum functions copied from the Linux kernel and this provides us with the same benefits as for the IPv4 header. For UDP checksum calculation a conceptual header is prefixed before the real UDP header containing the source address, destination address and protocol from the IPv4 header.

The checksum is calculated over the UDP header, pseudo-header, and the UDP data. RFC 768 specifies that the data is "padded with zero octets at the end (if necessary) to make a multiple of two octets.". In practice, we do this by making sure our UMEM frames are at least one byte larger than the maximum frame size and setting the byte after the last byte of our packet to zero if the amount of bytes of UDP data is uneven. This last step is essential even if initialize our UMEM with all zero bytes at the start of our program. Because we reuse the UMEM frames the byte following the new packet data might still contain data from an old, longer packet as we have seen in Section 2.5.3 and this byte of data can make our checksum invalid.

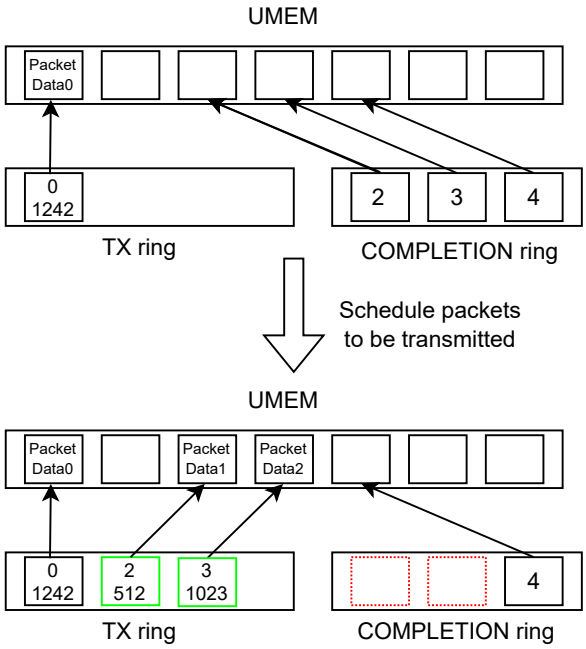

Figure 4.1: Sending more than one packet by requesting and submitting multiple packets at once from and to the COMPLETION and TX ring respectively

4.2.3 Receiving packets: AF_recvmsg()

The function definition is the following:

```
AF_recvmsg(struct AFXDPUDP_socket* sock, struct msghdr* message);
```

Compared to the *recvmsg()* system call described in Section 2.4.4 all the arguments are the same except for the file descriptor which we replace with a `AFXDPUPD_socket` pointer and the flags which we do not support. The reception of packets is done in the way described in Section 2.5.3: we grab the first descriptor from the RX ring, parse the headers in the UMEM, verify that they are a UDP header inside an IPv4 header inside an Ethernet header and verify the checksums of the IP and UDP headers. We copy the data into the provided buffer inside the `msghdr` and push the UMEM frame pointer to the FILL ring to be reused.

4.2.4 Batched sending of packets: AF_sendmmsg()

As an optimization to reduce the number of system calls, which we will experiment with due to them being expensive as we have seen in Section 2.3, we also support sending multiple packets at once, analogue to the *sendmmsg()* system call described in Section 2.4.5. To do so we use the same process as for sending a single packet, but instead of working with one UMEM frame (reference) at a time we can consume multiple at a time from the COMPLETION ring and submit multiple at a time to the TX ring. This is visualized in Figure 4.1 for a batch of two packets.

All the packet generation is still done packet per packet just like with the kernel's *sendmmsg()*. Compared to using *AF_sendmsg()* the differences are that we only do the TX and COMPLETION ring operations once per batch. We also do the *sendmsg()* system call once per batch instead of once per packet.

Ring name	Ring size
COMPLETION	2 * XSK_RING_CONS__DEFAULT_NUM_DESCS
FILL	2 * XSK_RING_PROD__DEFAULT_NUM_DESCS
RX	1 * XSK_RING_CONS__DEFAULT_NUM_DESCS
TX	2 * XSK_RING_PROD__DEFAULT_NUM_DESCS

Table 4.1: Default ring sizes

Name	Value
XSK_RING_CONS__DEFAULT_NUM_DESCS	2048
XSK_RING_PROD__DEFAULT_NUM_DESC	2048
XSK_UMEM__DEFAULT_FRAME_SIZE	4096

Table 4.2: Default constant values in our version of libxdp

4.2.5 UMEM layout and default buffer sizes

The default values for the ring sizes are shown in Table 4.1. In the UMEM we use a portion specifically for receiving and another portion specifically for sending, the portion for transmitting has the same amount of frames as the TX ring size, while the portion for receiving has the same amount of frames as the FILL ring size. The UMEM frames themselves are XSK_UMEM__DEFAULT_FRAME_SIZE long. We also stick with the aligned UMEM mode as the unaligned mode has a slight performance penalty as described in Section 2.5.3. All the default value constants on our system can be seen in Table 4.2.

The reason why we make the RX ring size smaller than all the other rings and half as small as the FILL ring is because of the reasoning in Section 2.5.3: we don't want the NIC to run out of new FILL ring pointers as this can have a large performance impact (we would rather run out of RX ring descriptors first). And we use aligned mode for the UMEM frames as it does not have a disadvantage for our use case and unaligned mode has slightly lower performance.

4.2.6 Netfilter alternative

As we have seen in Section 2.4.2 the kernel network stack implements NetFilter which provides firewall-like functionality as well as NAT and some more general functionality. As we have seen when describing the path through the kernel the Netfilter hook locations expect to be called in a specific location in the send or receive path and are tightly integrated with the kernel network stack functions and dataflow. Therefore utilizing Netfilter with AF_XDP is a non-trivial task.

However there is another way we can implement the functionality offered by Netfilter while using AF_XDP which is in the eBPF XDP program that runs when a packet arrives on the NIC (using NATIVE mode) or in the network stack (using SKB mode). Using eBPF we can inspect the packets that arrive, check for certain conditions, even modify the packet to do for example NAT. The work in Section 3.5.1 provides an IPTables implementation in eBPF.

Instead of implementing an entire firewall in the eBPF program, which is unnecessary for our tests and experiments we filter out a single type of packet as a proof-of-concept. The packet type of packet we filter out are ARP packets as we discuss in Section 4.2.7.

Another option is to handle the firewall in the user space application, as with zero-copy the cost of accessing the packet data in our user space application is minimal we can run our firewall here. We implement a very limited "firewall" in our application which drops any non-UDP and non-IPv4 packets. It does so by checking the protocol numbers: if the Ethertype inside the Ethernet header does not equal IPv4 or the transport layer protocol number inside the IPv4 header does not equal UDP we simply drop the packet without processing it.

A major disadvantage to this approach is that every application needs to implement its own firewall or filters, while for Netfilter the filter rules are applied system-wide.

4.2.7 ARP

We provide a way to use the application without the Linux kernel ARP implementation. We do this by creating an in-memory map that contains IPv4 addresses as key and MAC addresses as value. As the server never initiates the connection we rely on incoming packets from which we extract the IPv4 and MAC addresses and add them to the map for later use when responding. For the client which does initiate the connection we allow the user to pass command line arguments that specify the IPv4 and MAC address of the server. However, this is not a scalable approach and also does not let mapping expire like ARP does.

Alternatively, we can keep using the Linux kernel ARP functionality by using eBPF. If we filter out packets of the ARP Ethertype inside the Ethernet header and instead of forwarding them to our AF_XDP socket we pass it to the kernel using the XDP_PASS as we have seen in Section 2.5.2. The section of code we use for this looks as follows:

```
// Assume the packet starts with an Ethernet header
struct ethhdr *eth = packetdata;
__u16 h_proto;

// Confirm that the packet is at least the length of an Ethernet header, if not drop it
if (packetdata + sizeof(struct ethhdr) > packetdata_end)
    return XDP_DROP;

// If the protocol in the Ethernet header is ARP, pass the packet to the kernel network stack
if (eth->h_proto == htons(ETH_P_ARP))
    return XDP_PASS;

// If the packet is a non-ARP packet forward it to our AF\_XDP socket
return bpf_redirect_map(&xsks_map, rx_queue_index, 0);
```

4.3 QUIC and HTTP/3

Instead of implementing the QUIC protocol ourselves, we can use an open-source implementation such as lsquic [Lit]. This provides us with the benefit of being able to focus on the underlying network stack as well as providing us with a tested solution that is also compatible with other QUIC implementations [See]. The choice for lsquic was made due to the fact that is written in C making it easy to integrate with our own C code.

We set the internal receive buffer size in lsquic to contain the same amount of packets as the size of the RX ring of our `AFXDPUDP_socket`.

4.3.1 HTTP server and client

The QUIC implementation that we use, namely lsquic comes with several example programs included. Two of them are a simple HTTP/3 server and client which utilize lsquic. We decide to modify these to support our own network stack as they are perfect for measuring the performance we can achieve over HTTP/3 and QUIC. We use the following features already provided by these programs:

- The server has a mode built in to serve a block of memory as file instead of reading an actual file. This eliminates the file I/O costs so we are sure that either QUIC or the network stack is the bottleneck.
- The client automatically generates statistics namely the average and standard deviation of: the time to connect, time per download and time to first byte. The client outputs

these statistics when enabled by a command line flag. We extend this to also output the exact download time per run to allow a more in-depth analysis.

Because of the way we designed our AFXDPUDP socket, the code modifications to convert from the kernel provided socket to our own were relatively simple. It comes down to the following steps:

1. Include our header: *#include "AFXDPUDP.h"*
2. Change the build system configuration or compile command to link our library
3. Remove *setsockopt()* calls because we do not support those, and use our `SocketOptions` struct for setting options
4. Change socket function call to include AF_ in front of them, for example *sendmsg()* becomes *AF_sendmsg()* and this for all functions.
5. For lsquic change the struct they use to save the different socket file descriptors to save a pointer to our `AFXDPUDP_socket` struct instead. In general: change the way the reference to the socket is changed from an integer file descriptor to a pointer to a `AFXDPUDP_socket` struct
6. Change any function calls that operate on the socket file descriptor, which we did not prepend AF_ to such as the *event_new()* call from libevent used by lsquic example programs to

These are the modifications we did to the lsquic example programs but they should be applicable to other C(++) QUIC implementations or even applications running plain UDP or another protocol on top of UDP.

4.4 Testing framework

While we are aware that a (QUIC) network testing framework already exists [Her+23] it works by setting up virtual networks using Docker. And while AF_XDP can work in such configurations [Fas], we chose to use physical NICs to get realistic measurements as our code does interact with the NIC driver (by waking up the driver) behaviour and performance might be different for virtual networks. Therefore we needed a simple way to run multiple tests with different parameters and over multiple versions to debug our software. During the development of our own framework another publication released theirs which has similar capabilities [Jae+23].

We used Ansible [Hata] to automatically push the code to the two test systems, compile it and run a variety of tests. Ansible is a relatively easy-to-use automation tool that can manage remote systems by utilizing ssh and python. In summary, after compiling has been done, we start the http_server on one system and then start the http_client on the other system, the http_client automatically exists when it is done with all its repetitions. If Ansible detects this, we let it kill the http_server on the other host and export the logs to our management computer. The management computer is connected to the test systems through a second NIC on both test systems.

To run the client and server Ansible starts bash script on the test systems, in the case that we are doing a test with specific hardware offloads enabled or disabled we enable/disable them as described in Section 2.2.1 which for example for checksumming gives us the following simplified view of a script:

```
1    sudo ethtool -K enp1s0f0 tx-checksumming off
2    sudo ethtool -K enp1s0f0 rx-checksumming off
3    run server
4    sudo ethtool -K enp1s0f0 tx-checksumming on
5    sudo ethtool -K enp1s0f0 rx-checksumming on
```

Chapter 5

Experiments and results

In this chapter we present our measurement results for both our own AF_XDP UDP socket as well as the normal kernel UDP socket. We begin by describing our test setup, the exact hardware used, and some software details of importance. After that we introduce a naming convention to be able to use abbreviations for the rest of the chapter. We also look at a method we use to gauge the CPU efficiency of the network stack and last but not least we look at our experiments and their results.

5.1 Setup

For our tests we have 2 test systems:

Test System 1:
- CPU: Intel Core i3-4160
- RAM: 8GB dual channel
- OS: Ubuntu 22.10
- kernel: 5.19.0-46-generic
- IP: 192.168.0.3

Test System 2:
- CPU: AMD A8-6500
- RAM: 7GB dual channel
- OS: Ubuntu 22.10
- kernel: 5.19.0-46-generic
- IP: 192.168.0.4

A diagram of the test setup can be seen in Figure 5.1. We use two Intel X540-T2 (10 GbE) NICs connected via PCI Express to the test systems. The choice for these NICs was made because of them having 10 gbps interfaces, because as Jaeger et al. [Jae+23] show: even without optimizations some QUIC implementations can reach about 3 gbps on modern CPUs, therefore 1 gbps NICs would be a bottleneck. Additionally the driver supplied with the kernel (ixgbe) supports AF_XDP (in driver mode) as well as zero-copy. The two NICs are connected using a single CAT 6A Ethernet cable of approximately 500 centimeters plugged into the first Ethernet ports. The systems themselves or their specifications were not chosen for a particular reason.

We do tests in two directions: meaning that both Test System 1 and Test System 2 act as HTTP server and HTTP client, we always mention which one is the server and which one is the client.

5.2 Software

As testing applications we use the http_server and http_client provided by lsquic which run HTTP/3, we modified them to support our AF_XDP socket as we explained in Chapter 4. To test the performance we use the "/file-2G" path provided by the server to serve a file of approximately 2 gibibyte (exactly 2147483742 bytes) from memory. We serve a file from memory to avoid (the cost) of file I/O, therefore file I/O does not have an impact on our performance.

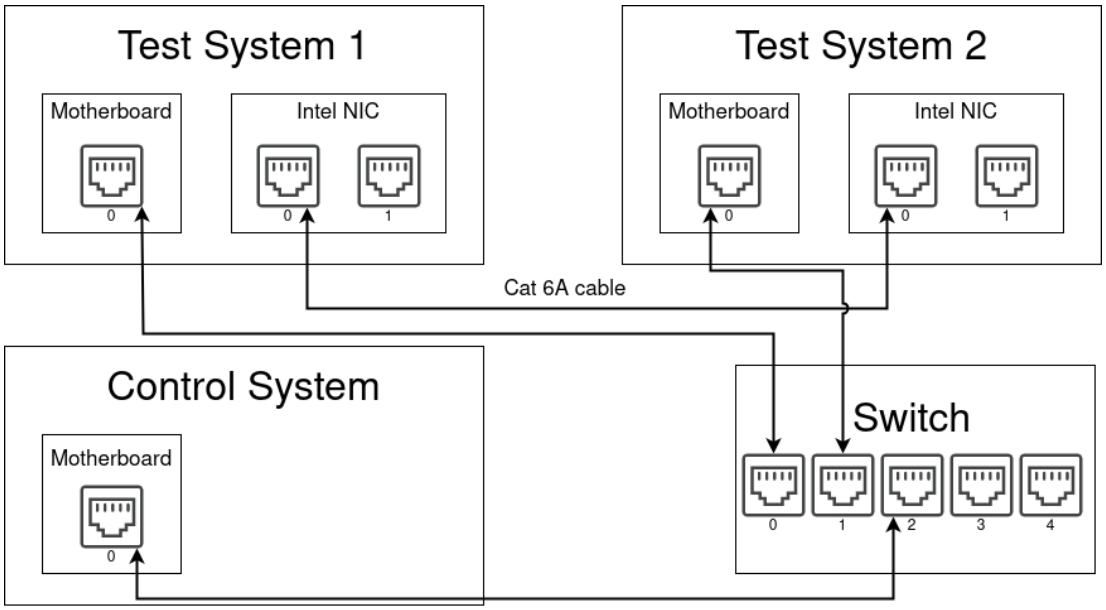

Figure 5.1: Diagram of the test setup

We calculate the goodput by timing how long it takes for the client to receive the entire 2 gibibyte file and then diving the amount of bits by the amount of seconds.

For the kernel network stack we test with and without checksumming, we do this by disabling/enabling both rx-checksumming-all and tx-checksumming-all on the interface using ethtool. For segmentation offloading (Generic Segmentation Offload and Generic Receive Offload) we disable/enable generic-receive-offload, generic-segmentation-offload and tx-udp-segmentation.

Unless mentioned otherwise we use CUBIC as congestion controller. To avoid the impacts of running the user space application and NIC driver on the same thread, as was explained in Section 2.4.5, we always run the user space application on another core than the NIC driver by using "taskset -c core_number command". We check which CPU handles interrupts with the method described in Section 2.2.2.

5.3 Naming convention

Because the configurations are long to explain over and over again we will use the following naming convention to refer to the configurations:

- AF_XDP_SKB: for AF_XDP in the default skb mode
- AF_XDP_NATIVE: for AF_XDP in native mode
- AF_XDP_ZC: for AF_XDP in native mode with zerocopy enabled
- KERN: for the Linux kernel network stack with default settings
- KERN_NOHWCSUM: same as KERN but with hardware checksumming disabled
- KERN_NOSEG: same as KERN but with segmentation offloads (GSO and GRO) disabled

Additionally, we append names to the end of these configurations to indicate further changes:

- MMSG_ followed by the maximum batch size to indicate that we are utilizing batches when transmitting and their maximum size. For example AF_XDP_ZC_MMSG_32 means that we use AF_XDP_ZC with batched sending and a maximum batch size of 32.

- CHECKSUM to indicate that calculating checksums is enabled, this is also the default meaning that if no checksum-related postfix is used we are alos using checksumming
- NOCHECKSUM to indicate that calculating checksums is disabled: both UDP and IPv4 checksums are not calculated
- NOIPCHECKSUM to indicate that we calculate the UDP checksum but not the IPv4 header checksum
- NOUDPCHECKSUM to indicate that we calculate the IPv4 header checksum but not the UDP checksum
- KERNLIKECHECKSUM to indicate that we simulate checksumming like the Linux kernel performs it by default without segmentation offloads, meaning calculating the IPv4 header checksum and UDP pseudo-header checksum in software and the rest of the UDP checksum in hardware, to simulate this we do not calculate the UDP checksum that is normally offloaded to hardware

5.4 Dividing the CPU's spent time into categories

To estimate where and how many CPU cycles are spent when running the server or client we utilize the same method as Jaeger et al. [Jae+23] which involves using perf to categorize functions and estimate how much time is spent in these categories. The categories we consider are:

- Crypto: functions that have to do with QUIC-related crypto calculations
- Uncategorized: For all functions we cannot assign a category to
- Packet I/O: functions related to moving packets through the network stack including costs like checksumming
- I/O: functions related to reading/writing files
- Connection management: functions that have to do with QUIC connection management

In case we use the default event the samples reported by perf are measured in CPU cycles [Fouf]. This means that it aligns with the number of clock cycles that were consumed by the CPU while executing the program or function being profiled. However, it's important to note that the number of CPU cycles required to execute a specific instruction or code segment can vary based on many factors, including the architecture of the CPU, the microarchitecture details, cache behavior, branch predictions, and more. As a result, the count of CPU cycles can serve as a relative measure of how efficiently the code is utilizing the CPU, rather than an absolute measure of time.

We extended the mapping from function names to categories provided by Jaeger et al. to include the functions used by our AF_XDP implementations. For the AF_XDP_SKB case, 0.37 % of the samples are uncategorized meaning that the impact of the functions we do not classify is negligible. While we could theoretically categorize every function, we chose not to as for some functions it's unclear where they belong exactly. For example *malloc()* is one of the functions generating uncategorized samples but it is used in multiple locations which means assigning it to a single category is incorrect.

However, this is only for the case where perf does not tell us where *malloc()* was called. In most cases perf also provides the function call history and if we for example see that *malloc()* is called within a function mapped to the packet I/O category we include the samples for that entire function in the packet I/O category, including the the CPU cycles of those specific *malloc()* and all other subfunctions calls.

We use a sampling rate of 1997 Hz, we don't use a round number such as 2000 Hz to prevent sampling in lockstep with periodic activity which could lead to misleading results [Gre].

When using the samples to compare implementations we use the amount of samples in a category relative to the total amount of samples. For example 51.85% of the samples of AF_XDP_SKB are spent on packet I/O. The reason why we do not use absolute number of samples is that the runs differ in length and therefore also in total number of samples, therefore one run might also have much more samples for packet I/O. This could for example come from a lot of retransmissions and not from an inefficient network stack. It would therefore be incorrect to directly compare the absolute number of perf samples for a category between two different runs and draw conclusions in terms of the efficiency of the processing in that category.

The reason why we don't simply measure the time *sendmsg()* and *recvmsg()* or *AF_sendmsg()* and *AF_recvmsg()* is that for receiving packets using the kernel network stack there no real processing happens inside the *sendmsg()* call. Only a memory copy from the receive buffer to user space as we have seen in Section 2.4.3. The same is true for the memory copy that happens for AF_XDP_SKB and AF_XPD_NATIVE which happens in the NAPI poll loop handler. For transmitting we have the same problem: not all processing is done in the function or subcalls: part of the processing is delayed or potentially offloaded to another CPU, namely the one running the NIC driver, by using a software interrupt. This means that any processing this interrupt handler does is not included in the time we measured. For example for AF_XDP adding the UMEM frames back to the COMPLETION ring to be reused can be considered part of the sending process but this happens in the software interrupt handler. Including the time until the interrupt handler is done processing the packet is also not a good idea as other tasks running inside the application or other processes can influence when the software interrupt is handled: this means another process can impact the performance we measure for the network stack.

We argue that taking perf samples over the entire run of the application is a good solution as we only measure the time that is being used for the application plus the network stack. We do not include the time spent waiting for software or hardware interrupts to be handled or the time that the CPU spends on another process. We do however include all the operations in the network stack including handling of software interrupts, this also means that we do not falsely classify a hypothetical modification that unknowingly causes less processing in user space but more processing in the software interrupt handler as more efficient.

5.5 Experiments and results

We begin with some general measurements to give us a general feeling of how the performance is for all the configurations. This also allows us to see if there are any outliers or unexpected results. After these general measurements, we zoom in on aspects that we expect to have significant performance impacts from our background knowledge from Chapter 2 as well as some things we noticed during our experiments.

5.5.1 General measurements

If we take Test System 1 as client and Test System 2 as server and iterate over the possible configurations we get the goodput results in Figure 5.2. We see a clear difference when the server is running AF_XDP_ZC: regardless of the client the performance is clearly the highest of all the server configurations. We do notice that Test System 2 is the bottleneck with the CPU consumption of the CPU core running the user space HTTP server at 100% while we saw varying CPU consumption on Test System 2 of about 60-75% on the CPU running the user space HTTP client depending on the test scenario.

Further, we notice that for both the client and server, disabling hardware checksumming offload and disabling segmentation offloads does not make a difference that can be considered significant, the average performance with all hardware offloads enabled is in many cases even slower than with them disabled. Although this is most likely caused by the small variance between measurements and not the hardware offloading being slower as the results are relatively close.

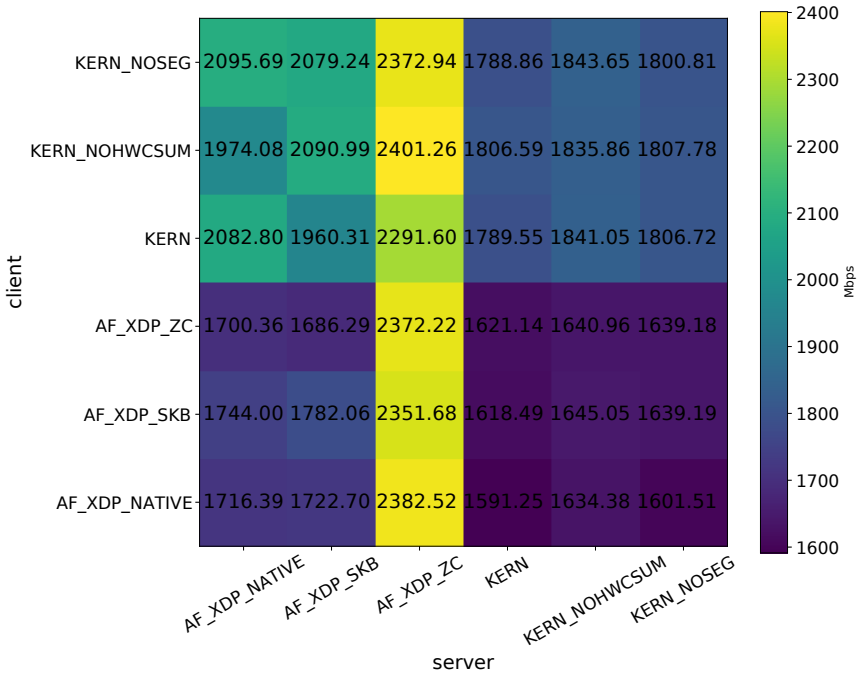

Figure 5.2: Heatmap of the goodput in Mbps of 10 repetitions with Test System 1 as client and Test System 2 as server

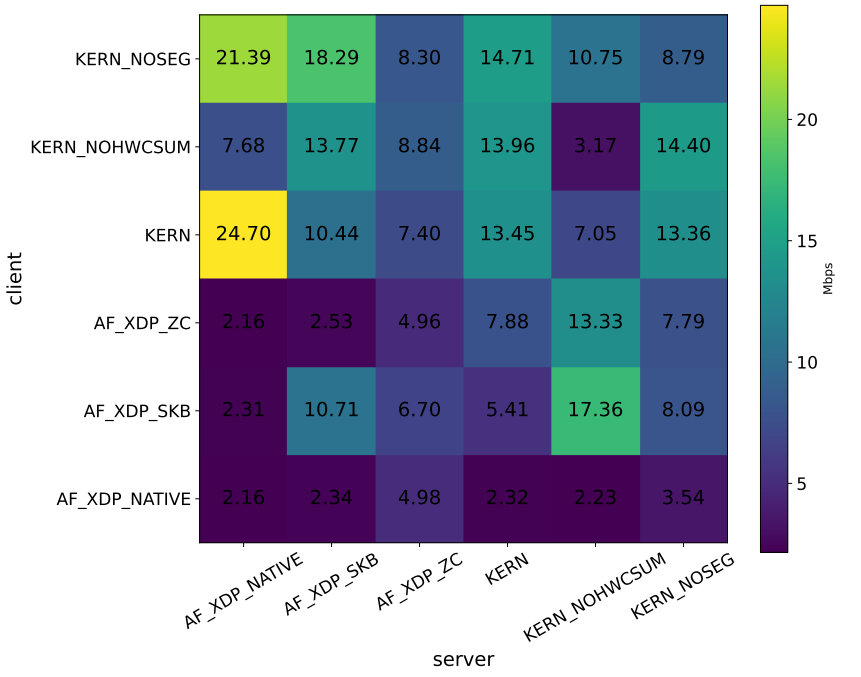

Figure 5.3: Heatmap of the standard deviation of the goodput in Mbps of 10 repetitions with Test System 1 as client and Test System 2 as server

For the server we can also conclude that there is no measurable performance improvement by using hardware offloads. Segmentation offloads not making a difference is consistent with the measurements published by Jaeger et al. [Jae+23]. From the heatmap we can conclude that

reducing the amount of copies in this test case definitely does improve the performance significantly as is evident from the fact that AF_XDP_ZC performs much better than AF_XDP_SKB and AF_XDP_NATIVE.

If we look at the heatmap plotting the standard deviations from these measurements in Figure 5.3 we can see that in general the standard deviation is relatively low indicating that from run to run there is little difference in the achieved goodput. We conclude that there is no unexpected behavior in terms of variance between the runs.

Inverse scenario

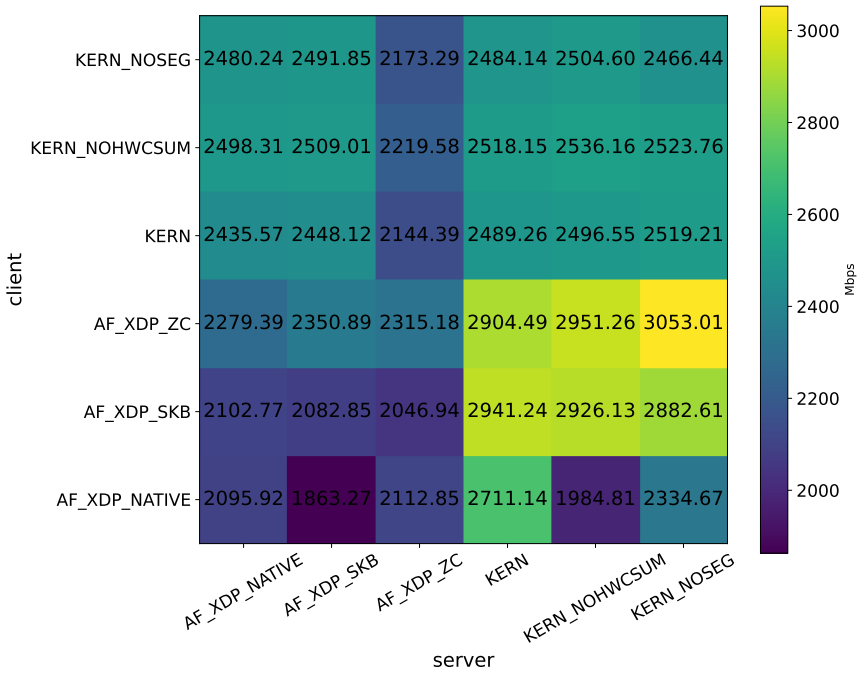

Figure 5.4: Heatmap of the average goodput in Mbps of 10 repetitions with Test System 2 as client and Test System 1 as server

If we invert the roles and make Test System 1 the server and Test System 2 we get the results that can be seen in Figure 5.4, we do again notice that Test System 2 is the bottleneck with a CPU consumption of 100% while that of Test System 1 stays below 100% all of the time. Unlike our first heatmap, we do not have a single row or column that clearly has higher values than the others: there is no configuration that is the fastest in all scenarios. But we do notice that the highest performance is achieved by using AF_XDP on the client and the kernel network stack on the server. In this case, there does not seem to be a significant difference between using zero-copy mode or not on the client. In terms of standard deviation which can be seen in Figure 5.5 we do not notice anything noteworthy.

As we have noticed by now, Test System 2 is the slowest of the two systems, therefore from now on we will mainly use this node when trying to measure the impact of our optimizations. This is because improving performance on the non-bottleneck node might not result in any performance improvements. We do this specifically by changing the code or configuration on Test System 2 while keeping the code or configuration on Test System 1 constant. In situations where we are trying to improve receive bottlenecks we will run the client on Test System 2 and in situations where we are trying to improve transmit performance, we will run the server on Test System 2. However, we always mention which software configurations the nodes run and which node runs the server and which one runs the client.

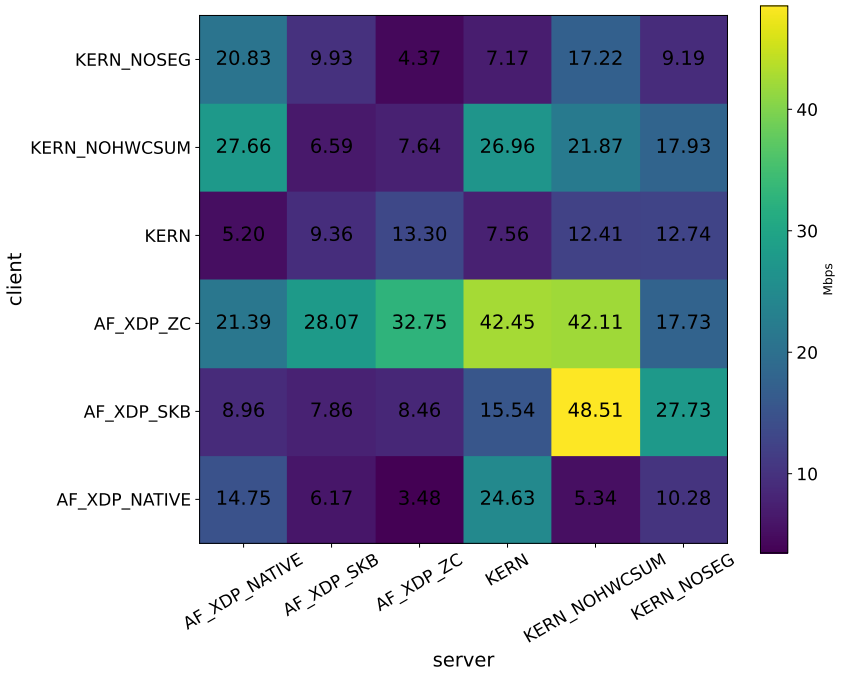

Figure 5.5: Heatmap of the standard deviation of the average goodput in Mbps of 10 repetitions with Test System 2 as client and Test System 1 as server

5.5.2 Memory copies

As we get a great performance improvement when running the server on Test System 2 using AF_XDP_ZC we also want to see if this makes it so the CPU spends relatively less time on packet I/O using the method described in Section 5.4. We visualize this for 10 repetitions with Test System 1 as client using AF_XDP_ZC.

configuration	Crypto	Uncategorized	Packet I/O	I/O	Connection Management
KERN	98 279 933 973	889 935 616	223 685 189 640	70 123 993	39 645 657 652
AF_XDP_SKB	101 223 607 293	1 347 020 137	215 487 222 542	1 738 767	43 395 711 622
AF_XDP_NATIVE	99 601 602 651	1 277 885 572	211 333 339 171	4 602 733	41 407 722 170
AF_XDP_ZC	99 146 615 591	1 026 512 257	113 375 337 126	7 717 822	40 573 057 572

Table 5.1: Absolute number of samples from perf

As can be seen in Figure 5.6 our zero-copy version spends relatively less time doing packet I/O compared to the version using the kernel network stack and the AF_XDP versions that do make copies. The reason that the crypto and connection management bars become higher is that they now are relatively more expensive but in absolute numbers, we have about the same amount of samples as can be seen in Table 5.1. As removing a memory copy compared to AF_XDP_SKB and AF_XDP_NATIVE both improves performance significantly and also results in 46% fewer CPU cycles spent on packet I/O we conclude that reducing the number of memory copies has a great, positive impact on performance.

5.5.3 System calls

As we have seen in Section 2.3, system calls can be relatively expensive. And as we hypothesized in Chapter 1 we expect them to have a large impact on performance. Therefore we use strace to analyze the number of system calls made, because strace makes the program run much slower

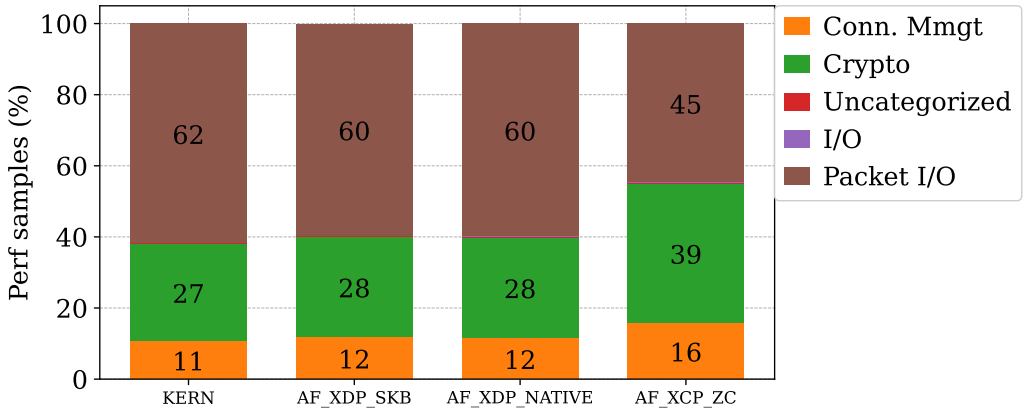

Figure 5.6: Comparison of the relative number of perf samples in each category compared to total, Uncategorized and I/O bars are so small they are (nearly) invisible and the number is not displayed: they contain less than 1% of the samples in all configurations.

we decided to limit the number of repetitions to five. On the client we used AF_XDP_ZC as configuration.

configuration	sendto	recvfrom	sendmsg	recvmsg
AF_XDP_SKB	7 475 059	0	0	14
AF_XDP_NATIVE	7 474 960	0	0	14
AF_XDP_ZC	7 475 163	7	0	14
KERN	0	0	6 092 380	2 738 390

Table 5.2: Amount of system calls on the server for five repetitions

configuration	packets received	packets sent
AF_XDP_SKB	7 474 970	3 351 537
AF_XDP_NATIVE	7 474 960	3 492 646
AF_XDP_ZC	7 474 983	3 714 282
KERN	6 092 359	2 740 231

Table 5.3: Amount of packets the client sent and received in the same test as in Table 5.2

We can see the results in terms of amount of system calls for the sending and receiving system calls used in Table 5.2. If we compare this to these numbers to the amount of packets received on the client in Table 5.3 we notice that that the amount of *sendto()* or *sendmsg()* calls on the server equals (almost) exactly the amount of packets received by the client. For the kernel network stack this is logical as for every sent packet a single *sendmsg()* system call is made. For the AF_XDP version however, we use the XDP_USE_NEED_WAKEUP flag which means that the NIC driver only needs to be woken up by a *sendto()* call if it signals so. As this flag was introduced to limit the number of times that we need to wake up the NIC we would expect to see a number lower than the amount of packets sent.

However, this might be caused by strace as it significantly slows down our user space program execution. While the NIC driver runs on another core and should not be impacted by strace, meaning that our application might be so slow compared to the NIC driver that the NIC driver can go to sleep between every transmitted packet while it might not have that opportunity (as

often) if our user space application is running at normal speed.

To check the amount of wakeup system calls with a normal execution speed we added a counter to the program to check how many times we need to wakeup for transmitting. The performance impact of this simple counter is negligible. The result was still always equal to or larger than the number of packets received by the client. Indicating that even at full speed the NIC driver still seems to go to sleep in between packet transmissions.

To gain a better understanding of this behavior we checked the NIC driver of our NICs and we see that it calls *xsk_set_tx_need_wakeup(pool)* if the XDP_USE_NEED_WAKEUP flag is set for every transmission being handled, this is done in the *ixgbe_clean_xdp_tx_irq()* function in /driver/net/intel/ixgbe_main.c. This means that with a batch size of one, there is no way to escape waking up the NIC driver for every packet transmission and there is no advantage to using the XDP_USE_NEED_WAKEUP flag for transmissions with the current ixgbe driver.

As we never came across this fact in any of the AF_XDP sources we consulted we dive a bit deeper and check if this is just a feature that our NICs lack or that there is never an advantage to utilizing the flag when transmitting. By searching for *"xsk_clear_tx_need_wakeup()"*, which is the function provided by XDP to clear the need_wakeup flag in the TX queue, in the Linux kernel (6.4) we do find exactly one driver utilizing it: the Melanox mlx5 driver. So only with (NICs using) this driver we expect a benefit when setting the XDP_USE_NEED_WAKEUP flag for transmitting. We will come back to this in our future work.

For receiving the ixgbe driver of our NICs does include logic to set or clear the flag, after receiving a frame in the NAPI poll mode it checks *"if (failure || rx_ring − > next_to_clean == rx_ring − > next_to_use)"*, which returns true if there is a failure or the hardware NIC RX ring is full. If this is the case it will signal that it needs a wakeup, because NAPI will stop polling and go to interrupt mode. If the check returns false the signal flag is cleared meaning that no wakeup is needed, but if there are packets left to process NAPI will keep polling. This logic explains our measurements of the number of *recvfrom()* system calls the AF_XDP runs make: all AF_XDP_SKB and AF_XDP_NATIVE do not make any and AF_XDP_ZC makes 7. We suspect this can be declared by the fact that the driver handles all necessary processing in the NAPI poll loop, therefore, we do not need to wake up the NIC driver unless the RX queue in main memory gets full, if this is the case it needs to be woken up so packets that are in the NICs RX queue can be transferred to the one in main memory: as these packets have already arrived they won't automatically generate a new interrupt if the software RX ring is cleared.

We can use strace to estimate the cost of each system call as can be seen in Table 5.4. We can estimate the time the program spends in the *sendto()* system call by multiplying for example for AF_XDP_SKB 10 μs times 7 475 059 calls, which would equal 75 seconds which is consistent with how long the server takes to run with strace enabled but without strace the runtime is smaller than the time it spents in syscalls with strace enabled (while the syscalls are the same). Therefore we conclude that strace introduces overhead to the system calls and we cannot trust its timing to reflect a "normal" use case.

As strace timing cannot be used we time how long *sendto()* calls take by calling *clock()* before and after every call, subtract the two results and we add all these time differences and also store the amount of calls. We get 0.8391 μs per call to *sendto()* as result while we make about 7 475 059 calls for five 2G files, which means we make about 1 495 012 calls per file which results in approximately 1.25 seconds per file spent in the *sendto()* system call. While the average 2G file download takes about 7 seconds, therefore we roughly spend 18% of the time processing the *sendto()* system calls, making it a good candidate to optimize.

5.5.4 Using AF_sendmmsg()

We can decrease the number of system calls made when transmitting by utilizing *AF_sendmmsg()* which reserves multiple UMEM locations, and fills them with all the packets passed, meaning

configuration	sendto	recvfrom	sendmsg	recvmsg
AF_XDP_SKB	10 μs	N/A	N/A	13 μs
AF_XDP_NATIVE	10 μs	N/A	N/A	14 μs
AF_XDP_ZC	5 μs	7 μs	N/A	11 μs
KERN	N/A	N/A	13 μs	8 μs

Table 5.4: Average time spent per system call for five repetitions according to strace

configuration	sendto	recvfrom	sendmsg	recvmsg
AF_XDP_ZC	7 475 163	7	0	14
AF_XDP_ZC_MMSG	104 538	0	0	14

Table 5.5: Amount of system calls on the server for five repetitions (with maximum batch size equal to 1024)

that one system call is made for an entire batch of packets instead of one per packet. By doing this we can drastically reduce the amount of *sendto()* system calls made as can be seen in Table 5.5. Our version using *AF_sendmmsg()* uses 71 times less *sendto()* system calls, when utilizing a maximum batch size of 1024 in lsquic, compared to processing packets one at a time.

We use Test System 2 as server and Test System 1 as client, meaning that in normal operation with both systems running AF_XDP_ZC Test System 2 is the bottleneck with the CPU core at 100% while Test System 1 peaks at about 73%. And since we do 7370625 fewer *sendto()* calls (over 5 files) which we estimated at 0.8391 μ seconds per call we expect to be able to save (7370625/5) * 0.8391 μs = 1,2369 seconds per file.

configuration	average time (ms)	std of time (ms)	average goodput (Mbps)	std of goodput (Mpbs)
AF_XDP_ZC_MMSG_1024	7509.69	10.11	2287.69	3.08
AF_XDP_ZC_MMSG_128	6153.41	28.91	2791.93	13.14
AF_XDP_ZC_MMSG_64	5916.05	46.03	2903.94	22.63
AF_XDP_ZC_MMSG_32	5912.60	32.86	2905.64	16.17
AF_XDP_ZC_MMSG_16	5982.84	29.51	2871.52	14.19
AF_XDP_ZC_MMSG_8	6107.67	26.94	2812.84	12.43

Table 5.6: Performance with different batch sizes for 50 runs with client using AF_XDP_ZC and a send buffer of 2 times the default

However this expected performance improvement does not happen in reality if we use the default settings. When we log the batch sizes being sent by the server we saw most of them hit the maximum allowed by the default maximum lsquic batch size which is 1024 packets. The performance we measure is even worse than without batches. If we however decrease the batch size we notice that we do see a performance improvement, as can be seen in Table 5.6. To achieve these results we also increased to send buffer size, we will explain why in Section 5.5.7. We achieve the optimal performance at a batch size of about 32, this does limit our reduction in system calls to a maximum of 32 times, instead of the 112 times fewer system calls we saw with a batch size of 1024.

We suspect that this behavior is caused by the larger delay between a packet being generated and actually being submitted to the TX ring with increased batch sizes. Therefore, the larger the batch size the longer it takes for the first few packets in the batch to be transmitted. This could potentially have a negative impact on things like congestion control.

But we do still see a reduction in relative CPU cycles spent for Packet I/O as can be seen in Figure 5.7 we also spend relatively less CPU cycles on Packet I/O leading us to the conclusion

that reducing the number of system calls has a positive effect on both real-world goodput and CPU cycles spent.

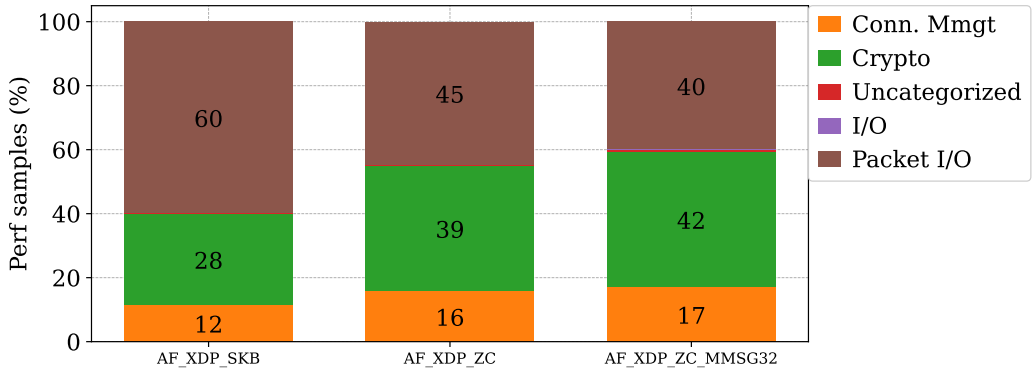

Figure 5.7: Comparison of the relative number of perf samples in each category compared to total, Uncategorized and I/O bars are so small they are (nearly) invisible and the number is not displayed: they contain less than 1% of the samples in all configurations.

The batched sending method we have tested here for our AF_XDP implementation is based on the same optimization available in the Linux kernel namely the *sendmmsg()* system call that we described in Section 2.4.5]. Next we look into this and check if we achieve similar results with the Linux kernel network stack.

5.5.5 Using sendmmsg()

Lsquic also allows, for the kernel network stack version, the usage of *sendmmsg()* as well as *recvmmsg()*. The latter provides the same optimization but for receiving. However out of the box enabling both does not seem to work correctly on the systems we tested this on and Jaeger et al. [Jae+23] mentions experiencing the same problem and therefore does not report results with these options enabled. But we did notice that enabling just *sendmmsg()* does work. We benchmark the performance impact of using the kernel network stack with *sendmmsg()* for transmitting and the "standard" *recvmsg()* for receiving.

configuration	average time (ms)	std of time (ms)	average goodput (Mbps)	std of goodput (Mpbs)
KERN_SENDMSG	10746.88	143.75	1598.59	21.43
KERN_SENDMMSG_1024	9882.74	63.19	1738.37	11.13
KERN_SENDMMSG_128	8983.75	20.43	1912.33	4.36
KERN_SENDMMSG_64	8899.31	14.96	1930.47	3.25
KERN_SENDMMSG_32	8902.79	16.39	1929.72	3.56
KERN_SENDMMSG_16	9107.2	99.04	1886.41	20.55
KERN_SENDMMSG_8	9232.92	19.95	1860.72	4.03

Table 5.7: Performance for 10 runs with Test System 2 as server and the client using AF_XDP_ZC

As we see in Table 5.7 we get a similar result as with *AF_sendmmsg()* in terms of the improvement compared to not using batched sending. The optimal batch size seems to be in the same range with batch sizes 32 and 64 achieving the best results. This indicates that this performance improvement also maps well to the kernel network stack and is not AF_XDP or kernel-bypass

specific. This is what we expected as system calls have a relatively large cost regardless of the exact context or implementation they are used in.

5.5.6 Cost of checksumming

A feature that we did expect to provide a performance improvement but did not measure to actually provide one in our general measurements is hardware checksum offloading. We will now look further into this and see if we can explain why we did not measure a performance difference.

Our first step was to verify the results of the measurements we made: we used the method described in Section 2.2 to enable and disable checksumming and we can also utilize the method described there to verify that these features are enabled or disabled. But we are unaware of an easy-to-use method to verify that there is no kind of bug or incompatibility making it so hardware checksumming is turned on but not being used. Therefore, instead of trying to verify that hardware checksumming is functional, we utilize our AF_XDP network stack to estimate the cost of checksumming in software. Our AF_XDP network stack does not support hardware offloads but as the checksumming functions are located in our user space program we can easily time how long they take to execute.

We measure a cost per complete UDP checksum of 1.09726 μs by using *clock()*. Assuming about 1 500 000 packets being transmitted containing file data this would be about 1.635 seconds per file. To verify this we can try disabling checksum calculation completely and measuring the impact. This way we can measure the impact of not having checksum calculation at all on performance. Additionally, we can check the impact of the UDP and IP checksum individually by only disabling one.

configuration	average time (ms)	std of time (ms)	average goodput (Mbps)	std of goodput (Mpbs)
CHECKSUM	6024.10	27.77	2851.86	13.17
NOCHECKSUM	5182.98	26.80	3314.67	17.17
NOIPCHECKSUM	5872.98	16.00	2925.24	7.98
NOUDPCHECKSUM	5265.69	21.60	3262.61	13.40
KERNLIKECHECKSUM	5356.29	29.08	3207.42	17.44

Table 5.8: Performance for 10 runs with Test System 2 as server with AF_XDP_ZC_SENDMMSG32 as configuration and checksum calculation as shown in the table and the client using AF_XDP_ZC, both with send and receive buffers 8 times the normal size

configuration	average time (ms)	std of time (ms)	average goodput (Mbps)	std of goodput (Mpbs)
CHECKSUM	7271.23	31.92	2362.72	10.39
NOCHECKSUM	6752.51	17.20	2544.22	6.49

Table 5.9: Performance for 10 runs with Test System 2 as server with AF_XDP_ZC as configuration and the client using AF_XDP_ZC, both with send and receive buffers 8 times the normal size

What we see in Table 5.8 is that checksumming definitely does make a difference but not as much as the 1.635 seconds we estimated. We also increased the buffer sizes in these runs, we will come back to this in Section 5.5.7. As for KERNLIKECHECKSUM, the pseudo UDP header and IP header checksum calculations are relatively cheap compared to the UDP data (of MTU sized packets) but we do still see a noticeable effect in our measurements. This indicates that,

assuming all other conditions are right, work to move these checksums to hardware should have an impact on real-world performance.

If we compare checksumming versus no checksumming without batching we also measure a difference as can be seen in Table 5.9. As for the relative CPU cycles spent on Packet I/O relative to the other tasks we also see a difference by disabling checksumming as can be seen in Figure 5.8.

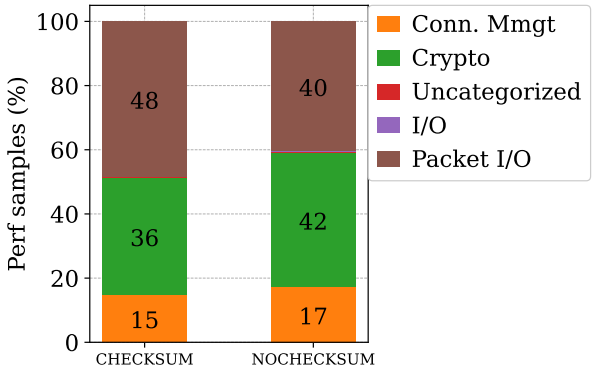

Figure 5.8: Comparison of the relative number of perf samples with checksum generation enabled disabled on the server (Test System 2) with both client and server using AF_XDP_ZC both with send and receive buffers 8 times the normal size. The number of Uncategorized and I/O samples is so small they are (nearly) invisible and the number is not displayed: they contain less than 1% of the samples in all configurations.

5.5.7 Buffer sizes

As we have mentioned a couple of times so far we have increased the buffer sizes for some of the experiments previously done, this is not without reason as we have noticed that without these increased buffer sizes performance was worse. We will now look into the effect buffer sizes have in this section. Both client and server have a receive buffer and send buffer size, this includes the RX, TX, FILL, and COMPLETION ring, as well as the lsquic internal incoming packet buffer.

		default buffer size		8 times default buffer size	
client	server	average goodput (Mpbs)	std of goodput (Mpbs)	average goodput (Mpbs)	std of goodput (Mpbs)
AF_XDP_ZC	AF_XDP_ZC	2372.22	4.96	2320.20	14.72
AF_XDP_SKB	AF_XDP_SKB	1782.06	10.71	1779.90	11.75
AF_XDP_NATIVE	AF_XDP_NATIVE	1716.39	2.16	1790.49	17.22

Table 5.10: Measurements using Test System 2 as server and Test System 1 as client

We start by doing our original tests, those without any extra optimizations, again with increased buffer sizes. We notice that we do not have significant measurable differences as can be seen in Table 5.10. Results for the other combinations of AF_XDP configurations not shown in the table also did not yield a difference. It's only once we enable batches and decrease the processing time per packet by disabling checksumming that we start to notice a difference.

To illustrate we compare some of the results from Section 5.5.6 where we took a look at the impact of checksumming and did increase the buffer sizes, to the results we obtained without

configuration	average time (ms)	std of time (ms)	average goodput (Mbps)	std of goodput (Mpbs)
CHECKSUM	7260.91	24.76	2366.08	8.08
CHECKSUM*	7271.23	31.92	2362.72	10.39
NOCHECKSUM	7535.34	93.21	2279.91	28.26
NOCHECKSUM*	6752.51	17.2.	2544.22	6.49

Table 5.11: Performance for 10 runs with Test System 2 as server, Test System 1 as client and both using AF_XDP_ZC.
* indicates that the buffer sizes were increased by 8 times

enlarging the buffers. As we can see in Table 5.11 the performance impact of checksumming does not show if we do not increase the buffer sizes.

We hypothesize that there are two possible reasons why the increased buffer sizes only have an impact in situations where we have already applied other optimizations:

- **The packet processing rate of the NIC versus that of the CPU**: the "small" buffers might only become a bottleneck once the number of packets the CPU can generate or process per second becomes high enough. For example for the server: if the NIC can transmit packets much faster than the CPU can generate them it's hard for the transmit buffers to be filled completely. But if we improve the packets per second that the CPU can output, we can also expect larger buffer utilization.
- **Bursts of packet transmissions and arrivals**:
  - On the transmit side: if we use batches a lot of packets are submitted to the TX ring at once, meaning that our TX buffer must be large enough to contain at least one batch size. But if the CPU can generate a second (smaller) batch of packets before the NIC has transmitted the first, the buffer needs to be even larger.
  - On the receive side bursts of packets arriving can cause the RX buffer to fill up: in our tests, Test System 1 has plenty of CPU power to process the packets sent by Test System 2 (that has a slower CPU, which is the bottleneck). And if the packet arrival times are spread evenly the CPU can handle the packets at a steady rate. If however batched sending is used, we can assume that the batched packets arrive in a small time window (in the best case scenario the NIC would achieve line rate while transmitting the entire batch) meaning that if the CPU in Test System 2 is not fast enough to handle packets at line rate, it needs to buffer the sudden influx of packets.

configuration	average time (ms)	std of time (ms)	average goodput (Mbps)	std of goodput (Mpbs)
NOCHECKSUM	7535.34	93.21	2279.91	28.26
NOCHECKSUM*	6752.51	17.20	2544.22	6.49
NOCHECKSUM**	6679.96	26.36	2571.85	10.16

Table 5.12: Performance for 10 runs with Test System 2 as server, Test System 1 as client and both using AF_XDP_ZC.
* indicates that the buffer sizes were increased by 8 times
** indicates that the transmit buffer size of the server as increased by 8 times

By testing the impact of each buffer individually we found out that the root cause of this improved performance is the increased transmit buffer size on the server. This can be seen in Table 5.12 where just increasing the transmit buffer size achieves the same performance as increasing all the buffer sizes.

Although our NIC is still much faster at transmitting packets than the CPU in Test System 2 is at generating them. This is because our NIC can hit the line rate of about 10 Gbps with MTU-sized packets while our CPU cannot. This means that theoretically if we have a TX ring (buffer) of for example 1024 packets and a maximum batch size of 512 and we start in a situation with an empty buffer, it would be impossible to fill the buffer completely: if the CPU submits an entire batch of 512 packets, 512 out of the 1024 slots in the buffer would now be full. The NIC can now start transmitting them while the CPU can start generating another batch of 512. And because the NIC is faster, once the CPU is done generating a new batch, the buffer is already back to empty. But this theoretical model might not be true in practice, as we've discussed the CPU running the network stack does a *sendto()* system call, which causes an interrupt request to wake up the NIC driver on another CPU, this wakeup will cost some time: it is done through a software interrupt so after it has been scheduled it might take until the next context switch before it is handled. The packet generation CPU has this delay as a headstart, further since some data structures are synchronized, therefore the NIC driver might sometimes have to wait for other tasks to complete, or it can be preempted [corbet] making it unable to send at full speed all of the time.

5.5.8 Interrupts

As we have seen in Section 2.2.2 and 2.3, hardware interrupts and software interrupts respectively can have a significant impact on performance. But we also saw that the core handling incoming packets is not necessarily the same as the one running the user space application in Section 2.4.5. So far when looking at CPU consumption we have only looked at the core running the user space application, in this section we will take a look at the core that is handling the interrupts and what happens if we run the user space application and interrupt handling on the same core. We do this in two distinct ways:

1. Run the user space HTTP client and the interrupt handler on different CPUs and measure the relative time the interrupt handling CPU spends handling the interrupts
2. Run the user space HTTP client and interrupt handling logic on the same CPU core and measure the performance impact compared to running on different CPUs

configuration	hardware interrupts (%)	software interrupts (%)
AF_XDP_SKB	0,00	10.57
AF_XDP_NATIVE	0,00	7.62
AF_XDP_ZC	0,00	0.29
KERN	0,00	23.41

Table 5.13: Relative CPU time spent on handling interrupts for the CPU that handles the NIC's hardware interrupts on the client. With Test System 1 as server using AF_XDP_ZC and Test System 2 as client, with the configuration in the table and 10 repetitions.

For 1 we can see the results in Table 5.13. It is clear that the different configurations have a significant difference in terms of relative time spent in software interrupts. We suspect the reason why the amount of time spent in hardware interrupts is virtually zero is because NAPI limits the amount of hardware interrupts by using polling. Using the method mentioned in Section 2.2.2 we look at the amount of hardware interrupts the NIC generates. Per repetition, about 40 000 hardware interrupts are generated while we receive about 1 500 000 packets for a 2G file. This indicates that NAPI works as we expect with our background knowledge from 2.2 and 2.4.3. So while a hardware interrupt is expensive due to the context switch, the number of interrupts that happen is very limited compared to the number of received packets and in addition, the hardware interrupt handler performs virtually zero processing. So we consider our measurement of virtually zero relative CPU time spent on hardware interrupts realistic.

The NAPI polling is executed inside a software interrupt and it is here that the actual processing is done. For AF_XDP_ZC the processing is virtually none, the only thing that needs to happen is to update the AF_XDP rings to indicate that a packet has arrived. For AF_XDP_NATIVE and AF_XDP_SKB the data also needs to be copied to the UMEM first, which is why we expect to see a larger utilization compared to AF_XDP_ZC and this is indeed what we measure. In SKB mode, the hook point is a bit later compared to NATIVE mode, and a `sk_buff` is constructed with the packet data which could explain the higher CPU consumption.

As for the KERN, inside the NAPI poll loop the packet is also pushed through the network stack as we have seen in Section 2.4.5. Thus the CPU utilization measured includes the processing for the network stack: Ethernet, Netfilter, IPv4 and UDP as well as creating a `sk_buff` but no memory copies happen.

	different core		same core		
Client	average goodput (Mpbs)	std of goodput (Mpbs)	average goodput (Mpbs)	std of goodput (Mpbs)	relative change in goodput (%)
AF_XDP_ZC	2272.29	12.61	2181.44	11.95	-4.00
AF_XDP_SKB	2136.91	29.82	1827.10	15.81	-14.50
AF_XDP_NATIVE	2139.24	31.86	1840.42	16.59	-13.97
KERN	2208.24	14.05	1922.84	13.06	-12.92

Table 5.14: Performance when running NIC driver and user space application on different cores vs running on the same core. Using Test System 1 as server as AF_XDP_ZC and Test System 2 as client with the configuration in the table.

For 2 we can see the results in Table 5.14, as we can see the performance impact with AF_XDP_ZC is minimal, this can be explained by having virtually no processing done in the NAPI poll loop. For AF_XDP_SKB and AF_XDP_NATIVE the results are as expected, but for the kernel network stack we would have expected the highest impact due to the high CPU utilization for software interrupts we saw before. But it seems like the operations the kernel network stack performs impact the performance less when running on the same core. We hypothesize this might be because AF_XDP_SKB and AF_XDP_NATIVE make a copy of the data in the NAPI loop while the kernel network stack does not.

5.5.9 All optimizations

In this section we take a look at the goodput and relative CPU samples collected by perf if we combine all the optimizations we discussed so far. This includes:

- Removing a memory copy by using AF_XDP zero-copy mode
- Using batched sending as described in Section 4.2.4, more specifically with a maximum batch size of 32, sending to decrease the number of system calls
- Disabling checksumming
- Increasing buffer sizes by 8 times compared to the default we described in Section 4.2.5

implementation	goodput (Mbps)
Kernel	1621.14
AF_XDP_NATIVE	1700.36
All Optimizations	3348.41

Table 5.15: Average goodput over 10 runs for different implementations running on Test System 2 as server and the client running AF_XDP_ZC for all

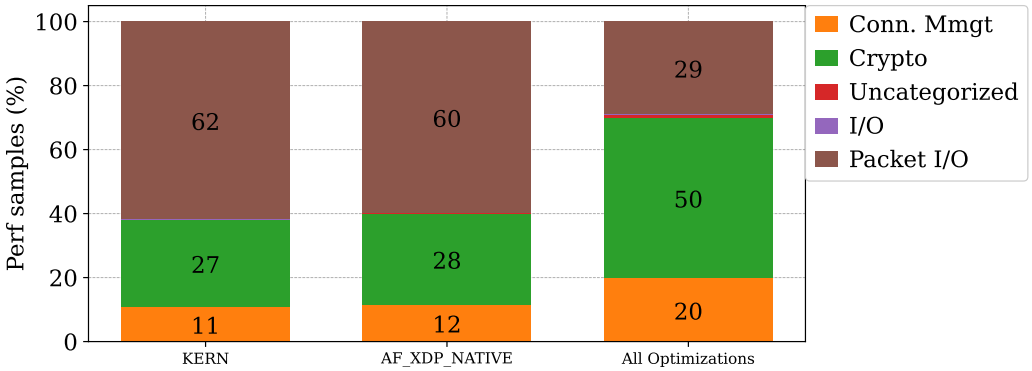

Figure 5.9: Relative number of perf samples spent in every category of the kernel network stack, our baseline AF_XDP_NATIVE implementation, and our implementation with all optimizations enabled. Uncategorized and I/O bars are so small they are (nearly) invisible and the number is not displayed: they contain less than 1% of the samples in all configurations except with all optimizations enabled they contain 1% of the samples.

The result in terms of perf samples can be seen in Figure 5.9, AF_XDP_NATIVE is our baseline as this is the configuration to which we added all the optimizations. As we can see we drop the time spent on the network stack (Packet I/O) from 60% to 29%. This also results in crypto-related operations relatively almost doubling from 28% to 50%. Therefore with all our optimizations, the crypto operations are now more expensive than the network stack. In terms of achieved goodput we see similar results in Table 5.15 with the goodput almost doubling with our optimizations.

Chapter 6

Conclusions

To conclude this thesis, we provide a summary of our final result, do a personal reflection and discuss potential future work.

Our final implementation is a UDP socket implemented utilizing AF_XDP to bypass the Linux kernel which we integrated into the HTTP/3 server and client included with lsquic to perform real-world HTTP/3 file transfer tests. As we kept our implementation simple and because all processing happens in user space, we could relatively easily tweak our implementation for a variety of experiments that we conducted. Additionally, due to the kernel bypass aspect, we could limit the system calls and memory copies.

In the introduction of this thesis, the research question we asked ourselves is which performance bottlenecks are present in the network stack that degrade QUIC performance. As a subquestion, we wondered if it is feasible to improve the performance of these bottlenecks and if these improvements have an impact on real-world performance.

We have successfully identified several bottlenecks with our experimental approach. The first bottleneck is memory copies. By utilizing zero-copy mode in AF_XDP and therefore avoiding a single memory copy we see a performance improvement of 40%. A second optimization is to reduce the number of system calls, because AF_XDP uses NAPI polling we already saw a very small number of system calls to receive packets. But to send packets we saw one system call per packet. To reduce this we used batched sending on the server which improved goodput by 22%. We also noticed that buffer sizes affected our performance and by doing multiple tests concluded that specifically, the size of the send buffer on the server was causing a bottleneck. By increasing its size we improved goodput by 12%. A last factor is checksum calculations, which we could not offload to hardware with AF_XDP so we simply disabled them improving goodput by 16%.

The combined effect of our optimizations is a goodput improvement of 97%. Although we have also shown that some of the bottlenecks are correlated: increasing buffer sizes does not have an effect until we have applied some other optimizations that improve the rate at which the CPU can send packets. This means that how large the performance improvement of an optimization is, also depends on the performance of other components. For example the faster we can get the CPU to send packets, the more we expect small buffer sizes to become a bottleneck and the larger the impact of increasing the buffer sizes becomes. Therefore we conclude from the goodput improvements we measured that the listed optimizations are significant in our tests and on our hardware. However the performance benefit might be smaller, nonexistent, or even greater in other scenarios and/or on other hardware. Although the optimizations we found were not novel and had already been described and implemented for other protocols. We did prove that they translate well to QUIC as is evident from the performance improvements we measured. Therefore, as there is nothing specific about QUIC that completely undermines these

optimizations, we expect them to also translate well to a variety of QUIC use cases as well as QUIC running on other hardware.

We also believe to have tackled the most restricting bottlenecks in terms of CPU utilization. This is evident if we compare the relative amount of CPU time spent on each of the core tasks of a QUIC implementation. As we have discussed in Section 5.5.9 the relative amount of CPU time spent on Packet I/O is 60% in our baseline and drops to 29% with all our optimizations.

As a result, the underlying network stack is no longer the most expensive operation for lsquic in terms of relative time spent: with all optimizations enabled crypto-related operations are more expensive than packet I/O. We have also shown that the CPU load on the CPU that handles incoming packet interrupts is reduced from 7.62% in our baseline AF_XDP_NATIVE test to 0.29% by enabling zero-copy mode. For reference: the kernel network stack has about 23.41% CPU consumption as the complete network stack processing of incoming packets happens on this CPU. While this processing happens on the CPU running the user space process in case of AF_XDP, we consider this an advantage when scaling to multiple parallel connections on a multi-core CPU.

As for our secondary research question, we have shown that part of the functionality that is lost, namely ARP and Netfilter can be implemented with AF_XDP. But while we have discussed traffic control and hardware offloads in the Linux kernel network stack, we have not provided these or an alternative in our implementation. This is because, as far as we are aware, it is currently not possible to use these with AF_XDP or provide alternatives that offer exactly the same functionality. We will discuss how this could be solved in our future work.

Personal reflection

This thesis deviated from its original idea significantly. The original idea, to look into implementing QUIC in the kernel to gain a performance benefit, was abandoned relatively fast. This is because, during the literature study, it became evident that the state of the art of network performance was not to move components into the kernel but rather to use kernel bypass solutions. Bypassing the kernel circumvents one of its inherent properties that negatively affects performance: the separation between kernel space and user space.

This separation for the network stack, NIC and NIC driver is avoided by kernel bypass solutions which, as we have shown leads to a large part of their performance benefit: decreased memory copies and system calls. However, I personally believe it is necessary to look critically at this, as the separation between user space and kernel space was put into place for a reason: security and stability. Due to a user space application not being able to access kernel space, it can not cause the entire kernel and therefore operating system to crash or go into an infinite loop. And although kernel bypass solutions were designed to mostly provide the same features, for example, eBPF which is used for AF_XDP sockets can run code in kernel space but in a safe manner as we have seen in Section 2.5. But it can cause the network interface to go down for the rest of the system: it is able to drop all incoming packets meaning that while errors in an eBPF program can't do harm to the kernel, it can prevent other user space applications from working correctly.

I believe that choosing for kernel bypass was not a wrong choice but I do believe that it is not suited for all scenarios. I believe that for end-user devices which typically run a variety of tasks and applications the kernel-provided separation between user space and kernel space is the most useful. While for (virtual) servers which typically run a single application such as an HTTP server, I do believe that the performance benefits outweigh the negatives.

As for my personal experience with this thesis: while I was pretty confident with my background research that AF_XDP could be used for what I intended, I am glad to say that it worked exactly as expected. There were some issues along the way: AF_XDP was more difficult to get working than expected, there were some bugs present once I integrated my socket in lsquic, some of which were very hard to squash as I initially did not get TLS decoding to work in Wireshark

and did not write any tests or checks. In hindsight, it might have been better to approach this in another way by creating some tests first or making sure I got TLS decoding to work so I could use the QUIC packet data to make sense of what was going on.

6.1 Future Work

In this section we will look into future work stemming from discoveries we made ourselves as well as work already described by others that we were unable to complete during this thesis.

6.1.1 Traffic control

A feature that we have not implemented in our own network stack is traffic control, this can be a major disadvantage when many streams are active as one (or more) could potentially overwhelm the others as we discussed in Section 2.4.2. But for our test scenarios consisting of a single stream of data the default fq_codel queuing discipline would not have made a difference as within a stream it queues packets in a first in first out way. While not trivial we do see several ways to get traffic control working using AF_XDP. In the scenario where all traffic originates from the same application but we have multiple sockets, we can use a shared UMEM: as we have seen in Section 2.5.3 a single UMEM can have multiple xsks associated with it. We could then run a scheduling thread and instead of submitting the completed packets to the TX ring associated with the UMEM we would submit them to queue managed by the scheduling thread, which runs a qdisc and submits the packets to the TX queue in the determined order. The method described here is similar to how the scheduler in DPDK works [Intb].

When running multiple applications that are not designed to cooperate this method would not work. The only possibility we see in this scenario is a traffic controller in hardware or using programmable NICs (smartNICs). The latter is the approach Xi et al. [XLW22] use to offload packet scheduling.

6.1.2 Hardware offloads in AF_XDP

In its current form it is impossible to run hardware offloads while using AF_XDP but as we prove in our measurements software checksumming can have a negative impact on performance. It would be beneficial to get hardware offloading working for AF_XDP. Therefore we see this as a way to make AF_XDP even faster in the future and as we have measured, the way the kernel network stack works without segmentation offloads is not optimal: the IP header checksum and UDP pseudo-header checksum are always calculated in software. And those do have a measurable impact on performance compared to not calculating the checksum at all so also in the kernel network stack better (more) hardware checksum offload support could improve performance.

For AF_XDP, hardware based XDP-hints [Jr17] might provide a way to implement hardware offloads in the future.

6.1.3 Test on other hardware

As we have seen in Section 5.5.3 the ixgbe NIC driver that we utilize does not have full support for the XPD_USE_NEED_WAKEUP for the transmit path. The Melanox mlx5 driver does seem to have better support for this. It would pique our curiosity to assess the performance implications of utilizing this flag by testing on such a NIC and driver. Additionally, it would be interesting to test on CPUs with multiple NUMA nodes, to see what the performance impact is if the CPUs have to share packet data across different NUMA Nodes. Or CPUs with direct cache support to see what its impact is.

6.1.4 Look into NIC drivers

As we discovered in Section 5.5.3 some NIC drivers contain logic for when they require the user space thread to wakeup the NIC driver when utilizing AF_XDP while others such as the ixgbe driver that we use do not. As far as we are aware there is no inherent reason why some drivers cannot support this. It would be interesting to see if logic can be added, if it improves performance and if the logic can be tweaked to increase performance even further.

6.1.5 Remove the last memory copy

We have shown the effect of reducing the amount of memory copies in Section 5.5.2. But we did not reduce the amount of copies in our implementation to zero. This is because we designed our implementation with the goal to easily replace the kernel-provided socket API. However as we have seen in Section 2.4.2 when calling *sendmsg()* the application needs to pass an allocated `msghdr`. The only way to get the data from the buffer referenced in this struct to a UMEM frame is by a memory copy. When utilizing AF_XDP zero-copy mode this is the only memory copy that still happens for packet data. However, it is not impossible to remove this last memory copy. The QUIC implementation just needs to be adapted to generate packets directly into the UMEM frames, for this a bit of management code is necessary as unlike the `msghdr`, the UMEM frames can't be directly reused after *sendmsg()* or in our case *AF_sendmsg()* has been called. We therefore need some additional logic in the QUIC implementation so it can consume UMEM frame pointer from (a wrapper around) the COMPLETION ring.

Bibliography

[1] *Linux manual page: packet (7)*. URL: https://man7.org/linux/man-pages/man7/packet.7.html (visited on 06/05/2023).

[2] *Linux manual page: syscalls (2)*. URL: https://man7.org/linux/man-pages/man2/syscalls.2.html (visited on 08/12/2023).

[3] *libxdp: library for attaching XDP programs and using AF_XDP sockets*. URL: https://www.mankier.com/3/libxdp (visited on 08/17/2023).

[App] Apple. *Apple Developer Documentation: QUIC Options*. URL: https://developer.apple.com/documentation/network/quic_options (visited on 08/13/2023).

[Ash] Satyadeep Ashwathnarayana. *Understanding Context Switching and Its Impact on System Performance*. URL: https://blog.netdata.cloud/understanding-context-switching-and-its-impact-on-system-performance/ (visited on 08/12/2023).

[Bau] Laura Bauman. *Harnessing the eBPF Verifier*. URL: https://blog.trailofbits.com/2023/01/19/ebpf-verifier-harness/ (visited on 08/07/2023).

[BC05] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

[BD17] Willem de Bruijn and Eric Dumazet. "sendmsg copy avoidance with MSG_ZEROCOPY". In: *The Technical Conference on Linux networking*. Vol. 2. 2017.

[Cai+21] Qizhe Cai et al. "Understanding Host Network Stack Overheads". In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. SIGCOMM '21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 65–77. ISBN: 9781450383837. DOI: 10.1145/3452296.3472888. URL: https://doi.org/10.1145/3452296.3472888.

[Car+16] Neal Cardwell et al. "BBR: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time". In: *Queue* 14.5 (2016), pp. 20–53.

[Cil] Cilium. *Cilium 1.15.0 developer documentation: Program Types*. URL: https://docs.cilium.io/en/latest/bpf/progtypes/#xdp (visited on 08/07/2023).

[Cor] Jonathan Corbet. *Page pinning and filesystems*. URL: https://lwn.net/Articles/894390/ (visited on 08/14/2023).

[corbet] corbet (username). *Driver porting: the preemptible kernel*. URL: https://lwn.net/Articles/22912/ (visited on 08/09/2023).

[CS18] Ruining Chen and Guoao Sun. "A Survey of Kernel-Bypass Techniques in Network Stack". In: *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*. CSAI '18. Shenzhen, China: Association for Computing Machinery, 2018, pp. 474–477. ISBN: 9781450366069. DOI: 10.1145/3297156.3297242. URL: https://doi.org/10.1145/3297156.3297242.

[Des] Mathieu Desnoyers. *The Linux Kernel documentation: Using the Linux Kernel Tracepoints*. URL: https://docs.kernel.org/trace/tracepoints.html (visited on 08/07/2023).

[eBP] eBPF. *What is eBPF? An Introduction and Deep Dive into the eBPF Technology*. URL: https://ebpf.io/what-is-ebpf/ (visited on 08/07/2023).

[Fas] John Fastabend. *XDP for virtio_net*. URL: https://lwn.net/Articles/708380/ (visited on 08/08/2023).

[Fle] Matt Fleming. *A thorough introduction to eBPF*. URL: https://lwn.net/Articles/740157/ (visited on 08/07/2023).

[Foua] Linux Foundation. *DPDK*. URL: https://www.dpdk.org/ (visited on 08/08/2023).

[Foub] Linux Foundation. *Linux manual page: IPv4 raw sockets (7)*. URL: https://linux.die.net/man/7/raw (visited on 06/05/2023).

[Fouc] Linux Foundation. *Linux source code (v6.4.9) - Bootlin*. URL: https://elixir.bootlin.com/linux/v6.4.9/source (visited on 08/12/2023).

[Foud] Linux Foundation. *Linux v6.4*. URL: https://github.com/torvalds/linux/tree/v6.4 (visited on 08/12/2023).

[Foue] Linux Foundation. *Linux wiki: GSO*. URL: https://wiki.linuxfoundation.org/networking/gso (visited on 06/20/2023).

[Fouf] Linux Foundation. *Sampling with perf record*. URL: https://perf.wiki.kernel.org/index.php/Tutorial#Sampling_with_perf_record (visited on 08/22/2023).

[Foug] Linux Foundation. *Software Interrupt Context: Softirqs and Tasklets*. URL: https://archive.kernel.org/oldlinux/htmldocs/kernel-hacking/basics-softirqs.html (visited on 08/12/2023).

[Fouh] Linux Foundation. *The Linux Kernel Archives*. URL: https://www.kernel.org/ (visited on 08/12/2023).

[Foui] Linux Foundation. *The Linux Kernel documentation: Checksum Offloads*. URL: https://docs.kernel.org/networking/checksum-offloads.html (visited on 08/12/2023).

[Fouj] Linux Foundation. *The Linux Kernel documentation: MSG_ZEROCOPY*. URL: https://www.kernel.org/doc/html/v4.15/networking/msg_zerocopy.html (visited on 08/14/2023).

[Fouk] Linux Foundation. *The Linux Kernel documentation: Packet MMAP*. URL: https://docs.kernel.org/networking/packet_mmap.html (visited on 06/05/2023).

[Gar+] Jim Garside et al. *COMP25111 - Knowledge Base Index*. URL: https://web.archive.org/web/20220511053226/https://xerxes.cs.manchester.ac.uk/comp251/kb/Context_Switching (visited on 08/12/2023).

[Gim] Sergio Gimenez. *Using netmap with TCP/IP application layer stack · Issue 711 · luigirizzo/netmap*. URL: https://github.com/luigirizzo/netmap/issues/711 (visited on 08/14/2023).

[Gre] Brendan Gregg. *Perf timed profiling*. URL: https://www.brendangregg.com/perf.html#TimedProfiling (visited on 08/22/2023).

[Gro] The Tcpdump Group. *libpcap code*. URL: https://github.com/the-tcpdump-group/libpcap/blob/master/pcap-linux.c (visited on 06/05/2023).

[Han+10] Sangjin Han et al. "PacketShader: A GPU-Accelerated Software Router". In: *SIGCOMM Comput. Commun. Rev.* 40.4 (Aug. 2010), pp. 195–206. ISSN: 0146-4833. DOI: 10.1145/1851275.1851207. URL: https://doi.org/10.1145/1851275.1851207.

[Hata] Red Hat. *Ansible*. URL: https://www.ansible.com/ (visited on 08/08/2023).

[Hatb] Red Hat. *Red Hat Customer Portal: Hardware Interrupts*. URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/reference_guide/chap-hardware_interrupts (visited on 08/12/2023).

[Hatc] Red Hat. *Red Hat Customer Portal: Receive Flow Steering (RFS) Red Hat Enterprise Linux 6*. URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/network-rfs (visited on 08/11/2023).

[Her+23] Joris Herbots et al. "Vegvisir: A Testing Framework for HTTP/3 Media Streaming". In: *Proceedings of the 14th Conference on ACM Multimedia Systems*. MMSys '23. Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 403–409. ISBN: 9798400701481. DOI: 10.1145/3587819.3592550. URL: https://doi.org/10.1145/3587819.3592550.

[HRX08] Sangtae Ha, Injong Rhee, and Lisong Xu. "CUBIC: a new TCP-friendly high-speed TCP variant". In: *ACM SIGOPS operating systems review* 42.5 (2008), pp. 64–74.

[IET] IETF. *RFC 9369 - QUIC Version 2*. URL: https://datatracker.ietf.org/doc/rfc9369/ (visited on 08/13/2023).

[IET80] IETF. *User Datagram Protocol*. RFC 768. Aug. 1980. DOI: 10.17487/RFC0768. URL: https://www.rfc-editor.org/info/rfc768.

[IET81] IETF. *Internet Protocol*. RFC 791. Sept. 1981. DOI: 10.17487/RFC0791. URL: https://www.rfc-editor.org/info/rfc791.

[Inta] Intel. *Data Plane Development Kit 23.07.0 documentation: AF_XDP Poll Mode Driver*. URL: https://doc.dpdk.org/guides/nics/af_xdp.html (visited on 08/14/2023).

[Intb] Intel. *Data Plane Development Kit 23.07.0 documentation: QoS Scheduler Sample Application*. URL: https://doc.dpdk.org/guides/sample_app_ug/qos_scheduler.html (visited on 08/15/2023).

[Intc] Intel. *Intel® Data Direct I/O Technology*. URL: https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html (visited on 08/12/2023).

[IT21] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. May 2021. DOI: 10.17487/RFC9000. URL: https://www.rfc-editor.org/info/rfc9000.

[Jae+23] Benedikt Jaeger et al. "QUIC on the Highway: Evaluating Performance on High-rate Links". In: *2023 IFIP Networking Conference (IFIP Networking)*. 2023, pp. 1–9. DOI: 10.23919/IFIPNetworking57963.2023.10186365.

[Jeo+14] Eun Young Jeong et al. "MTCP: A Highly Scalable User-Level TCP Stack for Multicore Systems". In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014, pp. 489–502. ISBN: 9781931971096.

[Jr17] Peter P. Waskiewicz Jr. "Accelerating XDP Programs Using HW-based Hints". In: (2017).

[Kar18] Magnus Karlsson. "The Path to DPDK Speeds for AF_XDP". In: 2018. URL: http://vger.kernel.org/lpc_net2018_talks/lpc18_paper_af_xdp_perf-v2.pdf.

[Laa] Kevin Laatz. *XDP unaligned chunk placement support*. URL: https://lwn.net/Articles/795014/ (visited on 08/07/2023).

[Lai+] Leonardo Lai et al. *UDPDK: A minimal UDP stack based on DPDK*. URL: https://github.com/leoll2/UDPDK (visited on 08/14/2023).

[Lan+17] Adam Langley et al. "The QUIC Transport Protocol: Design and Internet-Scale Deployment". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 183–196. ISBN: 9781450346535. DOI: 10.1145/3098822.3098842. URL: https://doi.org/10.1145/3098822.3098842.

[Lar] Michael Larabel. *Linux 5.1 Getting A Minor Spectre V2 Retpolines Optimization For Select Instances - Phoronix*. URL: https://www.phoronix.com/news/Linux-5.1-Retpoline-GCC-Opt (visited on 08/08/2023).

[LDS07] Chuanpeng Li, Chen Ding, and Kai Shen. "Quantifying the Cost of Context Switch". In: *Proceedings of the 2007 Workshop on Experimental Computer Science*. ExpCS '07. San Diego, California: Association for Computing Machinery, 2007, 2–es. ISBN: 9781595937513. DOI: 10.1145/1281700.1281702. URL: https://doi.org/10.1145/1281700.1281702.

[Lit] Litespeedtech. *lsquic: LiteSpeed QUIC and HTTP/3 Library*. URL: https://github.com/litespeedtech/lsquic (visited on 07/31/2023).

[Mara] Robin Marx. *HTTP/3 From A To Z: Core Concepts — Smashing Magazine*. URL: https://www.smashingmagazine.com/2021/08/http3-core-concepts-part1/ (visited on 08/07/2023).

[Marb] Robin Marx. *HTTP/3: Performance Improvements (Part 2) — Smashing Magazine*. URL: https://www.smashingmagazine.com/2021/08/http3-performance-improvements-part2/ (visited on 08/07/2023).

[Mar+20] Robin Marx et al. "Same Standards, Different Decisions: A Study of QUIC and HTTP/3 Implementation Diversity". In: *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. EPIQ '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 14–20. ISBN: 9781450380478. DOI: 10.1145/3405796.3405828. URL: https://doi.org/10.1145/3405796.3405828.

[Mia+19] Sebastiano Miano et al. "Securing Linux with a Faster and Scalable Iptables". In: *SIGCOMM Comput. Commun. Rev.* 49.3 (Nov. 2019), pp. 2–17. ISSN: 0146-4833. DOI: 10.1145/3371927.3371929. URL: https://doi.org/10.1145/3371927.3371929.

[Mica] Microsoft. *eBPF for Windows*. URL: https://github.com/microsoft/ebpf-for-windows (visited on 08/08/2023).

[Micb] Microsoft. *MsQuic documentation: platform support*. URL: https://github.com/microsoft/msquic/blob/9f74f69d0c16fadb62a332246daabac704bc7db0/docs/Platforms.md (visited on 08/13/2023).

[Micc] Microsoft. *Release XDP 1.0 for Windows*. URL: https://github.com/microsoft/xdp-for-windows/releases/tag/v1.0.0 (visited on 08/08/2023).

[Micd] Microsoft. *XDP for Windows forwarding sample*. URL: https://github.com/microsoft/xdp-for-windows/blob/v1.0.0/samples/xskfwd/xskfwd.c (visited on 08/08/2023).

[Net] Netfilter. *The netfilter/iptables project*. URL: https://www.netfilter.org/index.html (visited on 08/11/2023).

[NW13] David Ros Naeem Khademi and Michael Welzl. *Evaluating CoDel, FQ_CoDel and PIE: how good are they really?* 2013. URL: https://www.ietf.org/proceedings/88/slides/slides-88-iccrg-4.pdf (visited on 08/15/2023).

[Pir+20] Maxime Piraux et al. *QUIC Plugins*. Tech. rep. draft-piraux-quic-plugins-00. Work in Progress. Internet Engineering Task Force, Mar. 2020. 19 pp. URL: https://datatracker.ietf.org/doc/draft-piraux-quic-plugins/00/.

[Riz12] Luigi Rizzo. "netmap: A Novel Framework for Fast Packet I/O". In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, June 2012, pp. 101–112. ISBN: 978-931971-93-5. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo.

[See] Marten Seemann. *QUIC Interop Runner*. URL: https://interop.seemann.io/ (visited on 07/31/2023).

[Ser] Servermonkey. *Intel X540-T2 Dual Port 10GB Network Card*. URL: https://www.servermonkey.com/dell-intel-x540-t2-dual-port-10gbe-network-adapter.html (visited on 08/12/2023).

[Ten] Tencent. *F-Stack — High Performance Network Framework Based On DPDK*. URL: https://www.f-stack.org/ (visited on 08/14/2023).

[Tyu+22] Nikita Tyunyayev et al. "A High-Speed QUIC Implementation". In: *Proceedings of the 3rd International CoNEXT Student Workshop*. CoNEXT-SW '22. Rome, Italy: Association for Computing Machinery, 2022, pp. 20–22. ISBN: 9781450399371. DOI: 10.1145/3565477.3569154. URL: https://doi.org/10.1145/3565477.3569154.

[Wan+18] Peng Wang et al. "Implementation and Performance Evaluation of the QUIC Protocol in Linux Kernel". In: *Proceedings of the 21st ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*. MSWIM '18. Montreal, QC, Canada: Association for Computing Machinery, 2018, pp. 227–234. ISBN: 9781450359603. DOI: 10.1145/3242102.3242106. URL: https://doi.org/10.1145/3242102.3242106.

[WXW22] Minhu Wang, Mingwei Xu, and Jianping Wu. "Understanding I/O Direct Cache Access Performance for End Host Networking". In: *Proc. ACM Meas. Anal. Comput. Syst.* 6.1 (Feb. 2022). DOI: 10.1145/3508042. URL: https://doi.org/10.1145/3508042.

[xdp] xdp-project. *xdp-project bpf-examples: AF_XDP-example README*. URL: https://github.com/xdp-project/bpf-examples/blob/master/AF_XDP-example/README.org (visited on 08/07/2023).

[Xil] Xilinx. *OpenOnload high performance user-level network stack*. URL: `https://github.com/Xilinx-CNS/onload` (visited on 08/15/2023).

[XLW22] Shaoke Xi, Fuliang Li, and Xingwei Wang. "FlowValve: Packet Scheduling Offloaded on NP-based SmartNICs". In: *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. 2022, pp. 347–358. DOI: `10.1109/ICDCS54860.2022.00041`.