**BRNO UNIVERSITY OF TECHNOLOGY**
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

**FACULTY OF INFORMATION TECHNOLOGY**
**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

**DEPARTMENT OF COMPUTER SYSTEMS**
**ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ**


# XDP DRIVER FOR NDK PLATFORM

**XDP OVLADAČ PRO PLATFORMU NDK**

**BACHELOR'S THESIS**
**BAKALÁŘSKÁ PRÁCE**


**AUTHOR** **RICHARD HYROŠ**
**AUTOR PRÁCE**

**SUPERVISOR** **Ing. JIŘÍ MATOUŠEK, Ph.D.**
**VEDOUCÍ PRÁCE**



**BRNO 2024**

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

# Bachelor's Thesis Assignment

162799

| | |
|---|---|
| Institut: | Department of Computer Systems (DCSY) |
| Student: | **Hyroš Richard** |
| Programme: | Information Technology |
| Title: | **XDP Driver for NDK Platform** |
| Category: | Operating Systems |
| Academic year: | 2024/25 |

Assignment:

1. Study XDP interface of the Linux kernel. Consider the Linux driver part as well as the user application part.
2. Become acquainted with an interface for transmission of network packets between hardware and software parts of the NDK platform, which was developed by CESNET, and with software tools based on this interface.
3. Design an XDP driver for Linux kernel. Among others, focus on the possibility of its simple integration into the NDK platform.
4. Implement the designed XDP driver for DMA controllers utilized in the NDK platform.
5. Apply the implemented XDP driver in the software tools of the NDK platform.
6. Test the functionality and measure the parameters of the implemented XDP driver itself as well as the software tools of the NDK platform based on this driver.
7. Compare the measured parameter values of the implemented XDP driver with the selected alternative implementations.
8. Discuss the obtained results and possible future extensions of the thesis.

Literature:

- According to the supervisor's instructions.

Requirements for the semestral defence:

- Items 1 to 3.

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

| | |
|---|---|
| Supervisor: | **Matoušek Jiří, Ing., Ph.D.** |
| Consultant: | Ing. Martin Špinler |
| Head of Department: | Sekanina Lukáš, prof. Ing., Ph.D. |
| Beginning of work: | 1.11.2024 |
| Submission deadline: | 14.5.2025 |
| Approval date: | 31.10.2024 |

# Abstract

When trying to achieve high internet speeds, the CPU becomes a limiting factor. One of the most CPU-intensive parts of packet processing is sending packets through the Linux kernel network stack. Using this part of the operating system, which implements the L3 and L4 protocols and much more, is a trade of performance for convenience. That is why the technologies that circumvent the network stack emerged over time. These technologies usually take the whole packet processing to the user-space. In contrast, XDP (eXpress Data Path) is a technology that aims to achieve similar performance but still function as a part of the kernel itself by allowing the developer to decide early in the kernel network driver whether the packets should continue to the network stack, get filtered, or be redirected straight to the user-space. This driver implementation is the primary goal of this thesis.



# Abstrakt

Snahy dosáhnout vysokých internetových rychlostí vedou ke zjištění, že se CPU stává limitujícím prvkem. Jednou z nejnáročnějších částí zpracování paketů na CPU, je posílat je skrze network stack operačního systému. Tato část systému, která implementuje mimo jiné i L3 a L4 protokoly, představuje klasickou směnu rychlosti za pohodlnost. Z tohoto důvodu začaly vznikat technologie pro její obejití. Tyto technologie většinou přenáší celé zpracování do user-space. Toto je hlavní rozdíl XDP (eXpress Data Path), které se snaží dosáhnout stejného výkonu, ale stále fungovat v rámci kernelu. Dosahuje toho tím, že umožňuje vývojáři rozhodnout brzo v kernelovém ovladači, jestli bude paket poslán do network stacku, zahozen, nebo přesměrován přímo do user-space. Implementace tohoto ovladače je hlavním cílem této práce.



# Keywords

XDP, AF_XDP, kernel driver, packet processing, Linux

# Klíčová slova

XDP, AF_XDP, ovladač, zpracování paketů, Linux


# Reference

HYROŠ, Richard. *XDP Driver for NDK Platform.* Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jiří Matoušek, Ph.D.

# XDP Driver for NDK Platform

## Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Jiří Matoušek, Ph.D. The supplementary information was provided by Ing. Martin Špinler. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Richard Hyroš
May 12, 2025


## Acknowledgements

Thanks to Ing. Jiří Matoušek, Ph.D., for his help and patience with this bachelor thesis. Thanks to Ing. Martin Špinler for his help with the technical side of Linux kernel development and the intricacies of the NDK platform. Thanks to all my colleagues at CESNET who were always helpful in answering all the questions I had. And lastly, thanks to all my family who were rooting for me from the start to the finish.

# Contents


**1 Introduction** **4**

**2 The XDP Interface of the Linux Kernel** **5**
2.1 XDP – eXpress Data Path . . . . . 6
2.1.1 Typical Flow of Packets in the Linux Operating System . . . . . 6
2.1.2 Flow of Packets when XDP is Used . . . . . 7
2.1.3 XDP Program . . . . . 7
2.1.4 BPF Maps . . . . . 8
2.1.5 XDP Attach Modes . . . . . 8
2.2 AF_XDP . . . . . 9
2.2.1 UMEM . . . . . 9
2.2.2 Descriptor Rings . . . . . 9
2.2.3 XSKMAP . . . . . 10

**3 The NDK platform from a viewpoint of a network driver** **11**
3.1 Network Development Kit . . . . . 11
3.1.1 NDK DMA Controller . . . . . 12
3.1.2 Libnfb DMA Controller API . . . . . 12
3.1.3 DMA Medusa Descriptors . . . . . 12

**4 The design of the network driver** **13**
4.1 XDP and AF_XDP Operation . . . . . 13
4.2 Mapping the DMA Memory . . . . . 14
4.3 Page Pool API . . . . . 14
4.4 NAPI . . . . . 14

**5 Implementation of the Network Driver** **16**
5.1 The Driver Initialization . . . . . 16
5.1.1 Allocating the Net Device . . . . . 16
5.1.2 Setting up the Defaults of the Net Device Structure . . . . . 16
5.1.3 Initializing the DMA Controller . . . . . 16
5.1.4 Initializing the Carrier . . . . . 17
5.1.5 Network Device Kernel Hooks . . . . . 17
5.1.6 Initializing the XDP Functionality . . . . . 17
5.1.7 Finishing the Initialization . . . . . 18
5.1.8 Scheduling Peculiarity . . . . . 18
5.2 The Basic XDP Operation . . . . . 19
5.2.1 Initializing Page Pool . . . . . 19

5.2.2 Implemeting NAPI . . . 19
5.2.3 RX Path . . . 21
5.2.4 TX Path . . . 22
5.3 The AF_XDP Operation . . . 23
5.3.1 Mapping the Buffer Pool . . . 23
5.3.2 RX Path . . . 24
5.3.3 TX Path . . . 25

**6 The AF_XDP User-Space Applications 27**
6.1 Introduction to libxdp . . . 27
6.1.1 The Slow AF_XDP Fallback . . . 27
6.2 Structure of a libxdp AF_XDP Program . . . 28
6.2.1 Attaching the XDP Program . . . 28
6.2.2 Setting up UMEM with FILL and COMPLETION Rings . . . 28
6.2.3 Creating AF_XDP Socket with RX and TX Rings . . . 29
6.2.4 Mechanism for Counting Addresses . . . 29
6.2.5 RX Path . . . 29
6.2.6 TX Path . . . 29
6.3 Implementation to the NKD FPGA Platform User-Space Tools . . . 30
6.3.1 Common Parameters . . . 31
6.4 Mode ndp-xdp-read of ndp-tool Application . . . 31
6.5 Mode ndp-xdp-generate of ndp-tool Application . . . 31
6.5.1 Parameters . . . 31

**7 Testing 32**
7.1 Testing of the Default XDP Operation . . . 32
7.1.1 Tests Performed . . . 32
7.1.2 Conlcusion . . . 33
7.2 AF_XDP Benchmarks and Comparisons to NDP and DPDK . . . 33
7.2.1 Technologies Being Compared . . . 33
7.2.2 Testing TX Performance . . . 34
7.2.3 Conlcusion . . . 35
7.3 Testing RX Performance . . . 35
7.3.1 Conlcusion . . . 36

**8 Conclusion and future work 38**
8.1 Future Work . . . 38
8.1.1 Dynamic Adding and Deleting of the Linux Network Devices . . . 38
8.1.2 Support of Segmented Buffers in the XDP Network Driver . . . 38
8.1.3 Support Transmitting Segmented Buffers Redirected to the Driver . . . 39
8.1.4 Optimizations for the Basic Network Driver Operation . . . 39
8.1.5 Expand the ndp-tool with other XDP Modes . . . 39
8.1.6 Expand the ndp-tool XDP Modes with Hugepages Support . . . 39
8.1.7 Expand the ndp-tool XDP Modes with More Parameters . . . 39
8.1.8 Rework of Scheduling/Interrupt in the NDK FPGA Platform . . . 39

**Bibliography 41**

# List of Figures

2.1 XDP in the context of the Linux kernel network stack. By calling the XDP program early in the device driver, it can be decided what to do with each packet before it needs to be passed up to the network stack. Image sourced from [6]. . . . . 5
2.2 Simplified networking pipeline. Kernel driver forwards the packets to the operating system, concretely its part called kernel network stack. Application interacts with the network card through a socket. . . . . 6
2.3 Simplified networking pipeline with XDP program loaded at the start of the kernel driver makes a decision if a packet gets dropped, passed, or redirected. 7
2.4 A typical flow of XDP program execution. Image sourced from [6] . . . . . 8
2.5 Image shows UMEM and four types of descriptor rings. Image sourced from [7] . . . . . 9

3.1 NDK architecture, the driver primarily interacts with the DMA Module. Image sourced from [3] . . . . . 11

7.1 Comparison of different technologies, transmitting packets on single hardware DMA queue. . . . . 35
7.2 Comparison of different technologies, transmitting packets on single hardware DMA queue. . . . . 36

# Chapter 1

# Introduction

When trying to achieve high internet speeds, developers can look at many solutions, such as buying better hardware or optimizing their application, however these solutions can soon become very expensive or lead to diminishing returns. Another area where performance can be gained is by looking at how the packets are processed by the operating system. Developers often do not consider this area because, by design, the operating system is supposed to provide an abstraction layer that the developer uses to make development easier and faster. The part of the operating system that deals with packet processing is called the kernel network stack. Among other things, it implements the L3 and L4 protocols. Under heavy traffic load, the network stack becomes a limiting factor in the packet processing. That is why technologies to circumvent the network stack like DPDK (Data Plane Development Kit) were created, so developers who do not need the whole power of the network stack, but instead need to focus on processing a large amount of packets could trade the convenience of the operating system implementation for a solution that better suits their needs.
This is accomplished by using a user-space driver to bypass the stack altogether. In comparison, XDP (eXpress Data Path) aims to be equivalent to these technologies, accessible as an inbuilt part of the operating system. The way this works is by loading a specialized XDP program directly to the kernel at the earliest possible point in the packet processing path. This point lies in the kernel network driver, before any memory allocations are made. The program decides whether to drop the packet, send it to the network stack, or use the express path straight to the application running in the user-space. This way, developers can retain the advantages of the network stack, if they so desire, and use XDP as a sort of early prefilter. Otherwise, they can utilize the more performant solution and use the express path around the stack to get a similar performance to technologies that circumvent the stack, like DPDK, while still staying fully under the Linux system.
One of the disadvantages of the XDP is the required driver support for optimal performance. This support is the primary goal of this thesis.

# Chapter 2

# The XDP Interface of the Linux Kernel

This chapter describes the Linux kernel interfaces and mechanisms used while implementing the kernel driver together with a brief description of XDP and what it tries to achieve.

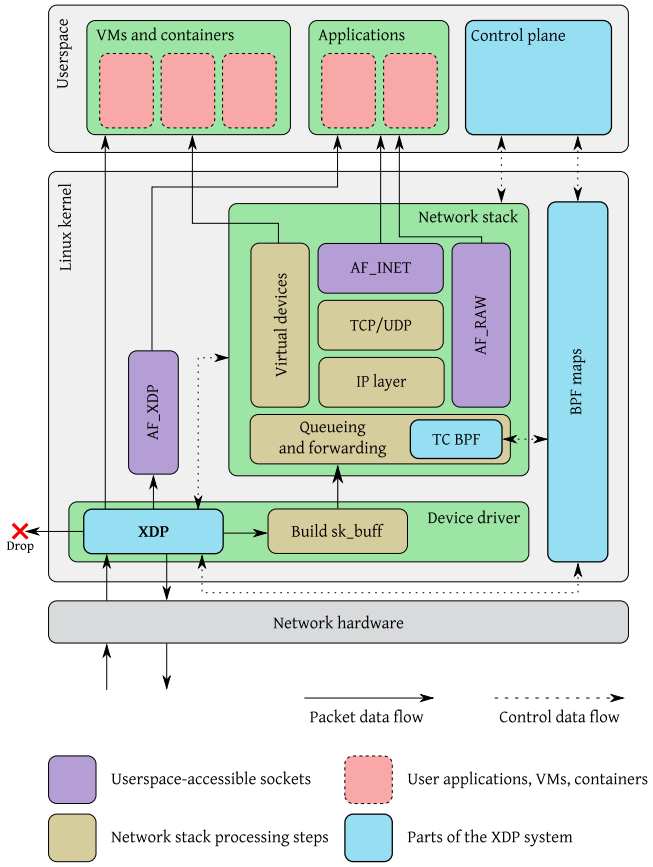


Figure 2.1: XDP in the context of the Linux kernel network stack. By calling the XDP program early in the device driver, it can be decided what to do with each packet before it needs to be passed up to the network stack. Image sourced from [6].

.

## 2.1 XDP – eXpress Data Path

The XDP framework is an in-kernel answer to various special-purpose toolkits (such as DPDK) for bypassing the operating system network stack in order to achieve high-speed packet processing. While other solutions usually bypass the operating system network stack entirely, XDP is instead incorporated early in the network driver data path by means of a hotswappable XDP program which is called after each received packet. This way, it can be decided what to do with the packet before any resource-intensive operations, such as buffer allocation, are carried out, and the user still retains all the benefits of using the network kernel stack [6]. XDP in the context of the stack is shown in Figure 2.1.

### 2.1.1 Typical Flow of Packets in the Linux Operating System

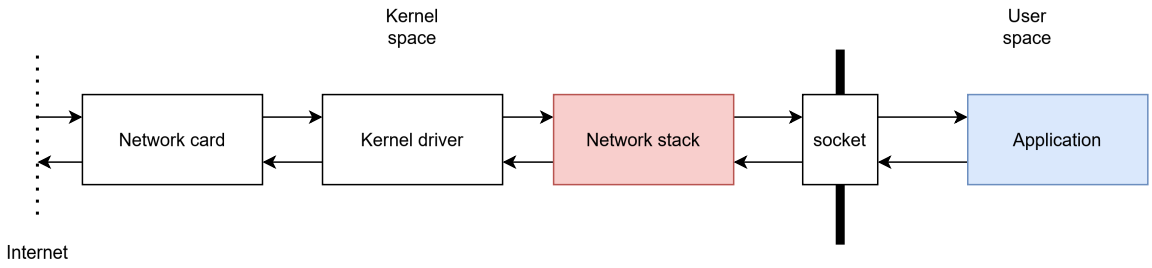


Figure 2.2: Simplified networking pipeline. Kernel driver forwards the packets to the operating system, concretely its part called kernel network stack. Application interacts with the network card through a socket.

.

To fully understand XDP, one must have at least a basic idea of how the Linux kernel handles packet processing. Figure 2.2 shows this simple packet pipeline. When the physical network card receives a packet, it needs a way to take this packet and give it to the operating system so that it can be processed and passed to the user-space application. The software that handles this task is called the kernel network driver. This driver runs in the kernel-space, which means that it has elevated access to memory and has no memory protection. An error in the driver will probably result in a crash. This is in contrast to the user application, which runs in the user-space with the memory protection of the operating system. The user application interacts with the network card through a socket and does not care about the implementation details of anything that comes before the socket.

### 2.1.2 Flow of Packets when XDP is Used

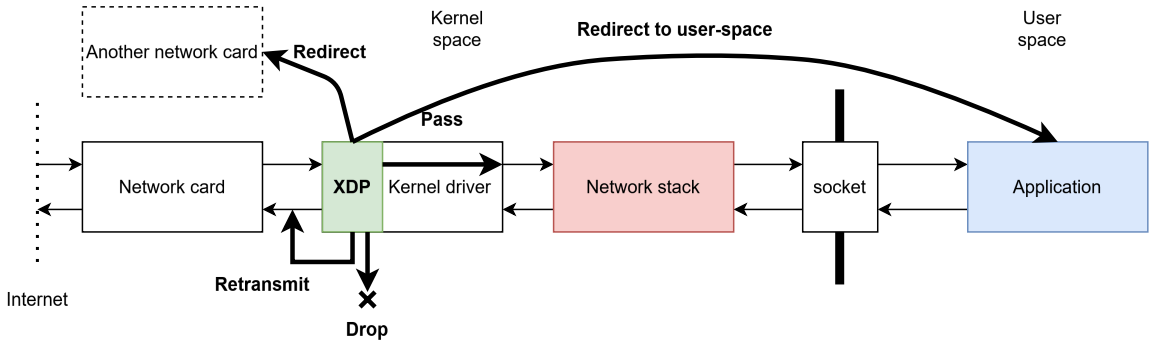


Figure 2.3: Simplified networking pipeline with XDP program loaded at the start of the kernel driver makes a decision if a packet gets dropped, passed, or redirected.

.

When XDP is used, the XDP program is loaded into the kernel driver, which decides what to do with each packet. This program can be run under kernel-space because it is compiled to instructions for an eBPF (extended Berkeley Packet Filter) virtual machine and statically checked for safety. Figure 2.3 shows the execution of the XDP program early in the pipeline and some of the actions that the packet can take instead of just continuing to the user-space through the network stack.

### 2.1.3 XDP Program

An XDP program is a special type of an eBPF (extended Berkeley Packet Filter) program that returns one out of the four possible return codes, each corresponding to a different XDP action:

- XDP_DROP – drops the packet – no further processing is done
- XDP_PASS – the packet is passed up the network stack
- XDP_TX – the packet is transmitted from the same interface it arrived on
- XDP_REDIRECT – the packet is either redirected for transmission to a different interface, to a different CPU as a part of a load balancing mechanism, or straight to the user-space via an AF_XDP socket, which bypasses the kernel network stack.

The XDP program is usually written in C using a helper library such as `libbpf`. It is then compiled into bytecode runnable by the eBPF virtual machine, which is part of the kernel. Before loading into the kernel, the XDP program is just-in-time compiled and statically verified to ensure that it does not cause corruption or a crash of the kernel.
The loading of the XDP program is usually done with the help of an external tool such as xdp-loader or straight from the C code using functions from the helper library like `libxdp`. The XDP program can be loaded and unloaded at any time during runtime, and multiple XDP programs can be loaded onto the same interface.
The starting point of the XDP program is a function with a struct containing a pointer to the raw packet data as its parameter. The XDP program has full read/write access to the data buffer, which allows for use cases such as filtering packets based on their content or

adding custom metadata to the packet [6]. The typical flow of the XDP program execution is depicted in Figure 2.4.

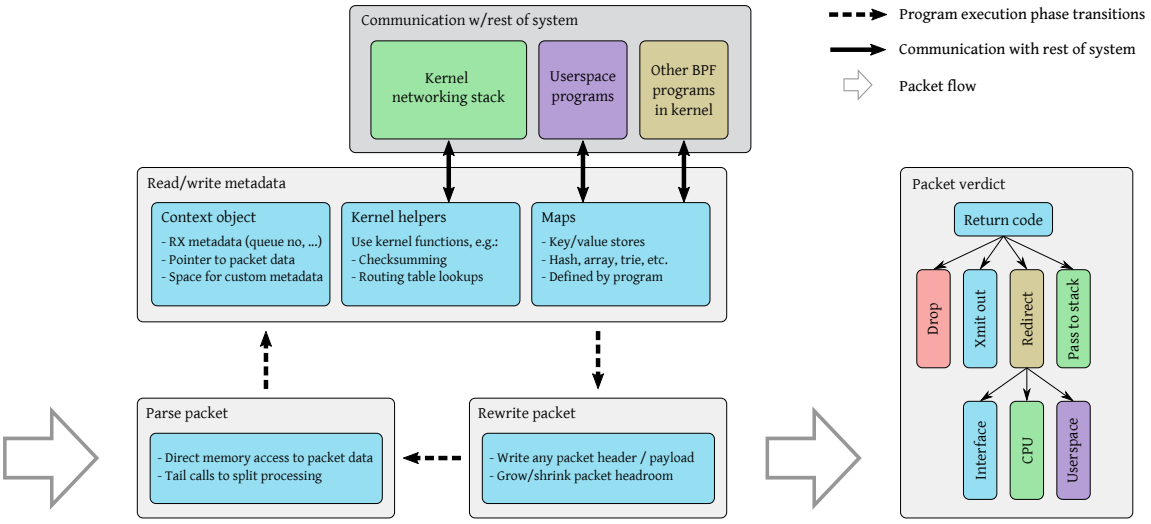


Figure 2.4: A typical flow of XDP program execution. Image sourced from [6]

### 2.1.4 BPF Maps

Maps are used to store data across XDP program invocations. They are structures of key/value pairs located in the kernel-space. Multiple XDP programs can share the same maps and the maps can also be accessed from the user-space via file descriptors. A typical use case of the map in the XDP context can be to count statistics about received packets. A special type of map called XSKMAP is also necessary to facilitate the redirection of packets to the user-space [1].

### 2.1.5 XDP Attach Modes

The XDP program can be loaded in three different modes with varying levels of support required by the hardware and kernel driver and varying levels of performance [1].

Native – the default mode, which is usually referred to when talking about XDP. The XDP program is run directly from the kernel driver on the receive path. Since the program invocation precedes the costly network stack operations and buffer allocations, the native mode offers increased performance.

Generic – this mode does not require any driver support. The XDP program is run further up the network stack, so it does not offer a substantial performance increase. This mode is instead meant to be used as a way to run XDP when the native mode is not supported or for testing purposes.

Offloaded – In offloaded mode, the program is loaded directly into a network card, freeing up the CPU from running the XDP program altogether, achieving a performance increase even in comparison to the native mode. This mode requires hardware support from the network card.

## 2.2 AF_XDP

AF_XDP is a socket type allowing zero-copy packet delivery between the user-space and the network card. In order to accomplish that, the XDP_REDIRECT operation is used. This means that the user-space application running AF_XDP needs to work in tandem with an XDP program that redirects traffic to the socket. This section describes how AF_XDP works and what needs to be done to use the AF_XDP socket – often referred to as XSK or xsocket.

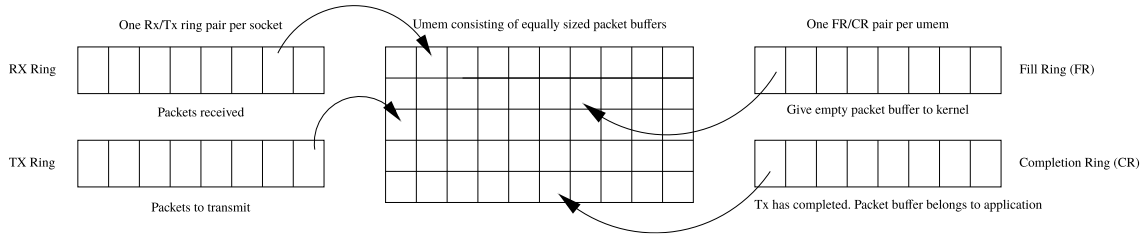


Figure 2.5: Image shows UMEM and four types of descriptor rings. Image sourced from [7]

### 2.2.1 UMEM

To achieve a zero-copy operation, the user-space application must first allocate a chunk of continuous memory called UMEM. This memory chunk is divided into equally-sized buffer cells, sometimes called frames. One such frame is going to hold a single raw packet together with a structure describing it. UMEM then needs to be registered with the socket.
Multiple sockets can share the same UMEM [5]. The UMEM and its four ring are shown in Figure 2.5.

### 2.2.2 Descriptor Rings

After registering the UMEM, there still needs to be some mechanism that allows for transferring ownership of UMEM frames between the user-space and the kernel-space. For this, AF_XDP uses single-producer/single-consumer rings. These rings hold descriptors, each pointing to one UMEM buffer.
There are four kinds of such rings. Each UMEM has two rings – these are called the FILL ring and the COMPLETION ring. Each socket has at least one of the two rings, called the RX ring and the TX ring, depending on the use of the RX queue, the TX queue, or both.

- The FILL ring is used for transferring ownership of empty user-space frames that are to be filled with packets by the card.
- The RX ring is used for transferring RX buffers back to the user-space after they have been filled out with received packet data.
- The TX ring is used for submitting filled out packet buffers to the card for transmission.
- The COMPLETION ring is used for collecting the frames back to the user application after the packet has been sent out.

The user-space application has the responsibility of tracking the used and unused descriptors and ensuring that the same descriptor is not enqueued onto multiple rings [5].

### 2.2.3 XSKMAP

Since the XDP program running in the receive path is what ultimately decides if a packet gets redirected to the user-space or not, there needs to be a way to distinguish between opened sockets. For this purpose, a special BPF map of type `BPF_MAP_TYPE_XSKMAP`, or for short XSKMAP exists. A reference to the AF_XDP socket is put into this map from the user-space application. The XDP program can then check if the received packet arrived on the queue opened by the socket and redirect the packet to it. Alternatively, this mechanism can also be used to balance traffic between multiple sockets [7].

# Chapter 3

# The NDK platform from a viewpoint of a network driver

This chapter describes the kernel driver interface from the side of the CESNET NDK (Network Development Kit) platform and how it influences driver design.

## 3.1 Network Development Kit

The Network Development Kit for FPGAs (NDK) is a framework designed for the development of FPGA-accelerated network applications. Framework's defining feature is the support of up to 400 Gbit ethernet speeds.

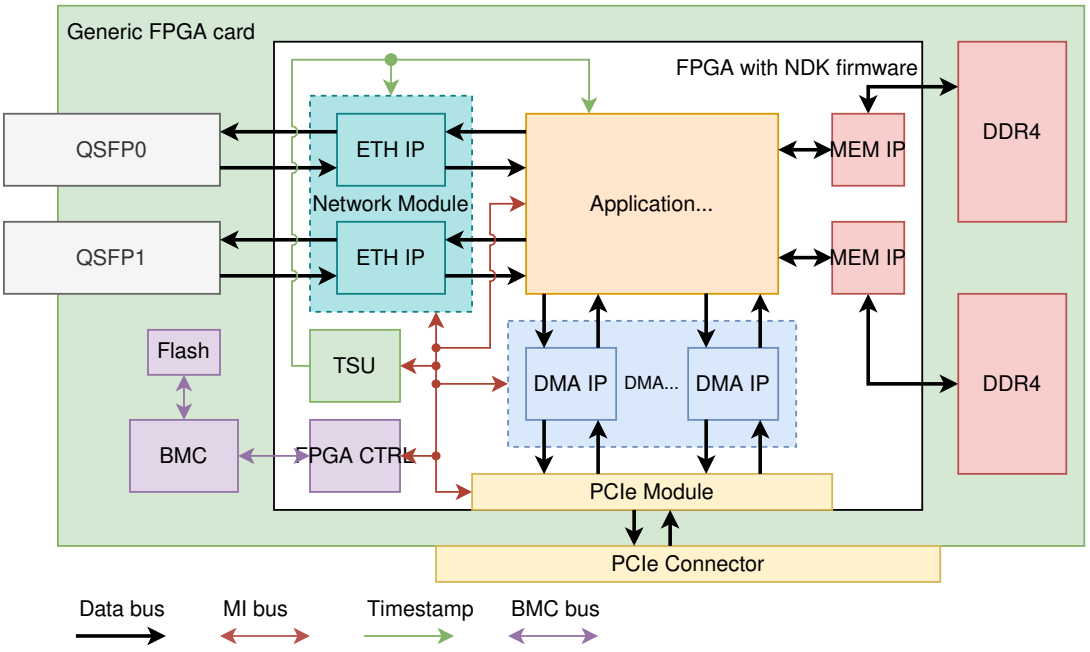


Figure 3.1: NDK architecture, the driver primarily interacts with the DMA Module. Image sourced from [3]

.

### 3.1.1 NDK DMA Controller

NDK achieves its high speeds using a DMA (direct memory access) mechanism. This allows for memory transfers from and to the network card without blocking the CPU. The NDK platform provides two implementations of DMA IPs: DMA Medusa (non-open-source module with speeds up to 400 Gbit) and DMA Calypte (open-source module under active development) [2]. Interaction with those DMA modules is possible through the libnfb library, which is part of the NDK software package NFB framework[1] [4]. The DMA modules are shown in Figure 3.1.

### 3.1.2 Libnfb DMA Controller API

The libnfb DMA controller API is located in the file `dma_ctrl_ndp.h`[2]. The kernel network driver uses this API to control the state of packet transfers by passing the ownership of the memory segments between the network card and the kernel driver. The ownership of memory is passed through the means of descriptors. At the start of the operation, the driver has to allocate a descriptor ring and a buffer that holds the descriptor counters. Those are the indices of the descriptor ring that point to the current frame being processed by either the driver (software descriptor pointer) or the network card (hardware descriptor pointer). If the network card's FPGA firmware has, for example, eight RX and eight TX queues, then this process will be done sixteen times, once for each queue.

### 3.1.3 DMA Medusa Descriptors

When the driver wants to transfer ownership of memory between the kernel and the network card, it does so by enqueueing the descriptor, which is filled with the buffer metadata, into the DMA queue descriptor buffer, and by updating the counters of the DMA queue. The descriptor is a 64-bit structure and DMA Medusa uses two kinds of these descriptors. In order to accommodate all of the packet metadata, such as packet data length, the most commonly used descriptor (type 2) can only hold the bottom 30 bits of a DMA address. The DMA controller works around this issue by using another type of descriptor (type 0), which holds the full address. When the transfer begins, the driver must first enqueue the descriptor of type 0 with the full DMA address of the first packet data. Then the driver can follow up with as many descriptors of type 2 – one packet per descriptor – as it wants as long as the upper bits of the DMA address do not change. If the address changes, the driver has to update it again by enqueueing the descriptor of type 0.

[1] https://github.com/CESNET/ndk-sw/
[2] https://github.com/CESNET/ndk-sw/blob/main/libnfb/include/netcope/dma_ctrl_ndp.h

# Chapter 4

# The design of the network driver

This chapter describes some of the kernel APIs and how they are used by the kernel network driver.

## 4.1 XDP and AF_XDP Operation

There is a difference between the kernel driver working in the basic XDP mode and the zero-copy AF_XDP mode.

In the basic XDP mode, the driver acts as a normal network driver, and the packets are passed to the kernel network stack. When the XDP program is loaded onto the interface, the driver executes it on each received packet and handles the result accordingly. In this mode, the driver allocates its own memory for the packet buffers. This mode only expands the basic driver functionality by running the XDP program.

In the AF_XDP mode, there is a user-space application that allocates the memory for the packet buffers, and the XDP program is used to redirect the packets into the user-space. This allows for zero-copy operation.

What this means for the driver is that when the switch between these modes happens, the affected queues – AF_XDP works with RX/TX queue pairs – have to be stopped and deinitialized so that the memory can be correctly freed by either driver when switching from XDP mode or the user-space when switching from AF_XDP mode, the queues then have to be rebuilt with correct memory model and restarted. This must happen on the fly at any time the user decides to open or close the AF_XDP socket.

## 4.2 Mapping the DMA Memory

The kernel normally uses virtual addresses, which have to be translated by the virtual memory system to CPU physical addresses using the memory management unit (MMU). IO devices, such as network cards, use the third kind of addresses – bus addresses. On some systems these can be the same as physical CPU addresses, but this is not true in general. Systems can have dedicated memory management units for IO devices (IOMMU). To use DMA, the kernel driver needs to therefore have a mechanism to create mappings between the virtual memory addresses (which the kernel driver uses to access the data) and the bus addresses (which the network card uses to access the data). This is accomplished through the kernel DMA API. Two types of such mappings exist:

The Consistent DMA mappings (also known as coherent) – these mappings are usually mapped at the driver initialization. Hardware should guarantee that the data can be accessed through these mappings by the CPU and the device in parallel without any explicit software flushing. In the network driver, they are used to map the descriptor rings through which the driver and the card transfer ownership of packets.

The Streaming DMA mappings – these mappings are usually mapped for one DMA transfer and unmapped as soon as possible. They need to be explicitly synchronized after the transfer is done. In the network driver, they are used to map the memory segments that contain packet data [8].

## 4.3 Page Pool API

The kernel Page Pool API is a convenient way to allocate and control memory for the use case of a network driver. It works by creating a pool of memory pages that the driver can then allocate. The API takes care of DMA mapping the pages, the pool holds the reference to the pages that are currently in use. The API is integrated into higher layers of the kernel network stack, meaning all the pages will be correctly released from the memory pool or recycled back into it after the page with a filled packet structure has been passed to the network stack. The API is directly supported by XDP by using the `MEM_TYPE_PAGE_POOL` memory model [10].

## 4.4 NAPI

The NAPI is a scheduling and event-handling mechanism used by the Linux kernel network stack. In the XDP mode, the driver creates a NAPI instance for each RX queue by calling the `netif_napi_add()` and passes to it a pointer to a NAPI poll loop, which is a function that takes care of the actual packet processing. From now on, the NAPI can be scheduled by calling the `napi_schedule()`. This needs to be done from an interrupt context or with disabled software interrupts. Since the NDK platform does not use hardware interrupts to signal the state of the DMA queues, the latter option was chosen for the driver.
While the NAPI poll loop is running, all interrupts are disabled [9]. This guarantees that the loop is tied to a single CPU and will not be rescheduled while running. Furthermore, the NAPI mechanism guarantees that there cannot be multiple instances of a single NAPI running at the same time. This allows for a lock-free RX queue and direct recycling of the page pool pages to its fast cache.

The XDP kernel API also cites that either NAPI or some other mechanism with the same guarantees is needed when working with redirection to the user-space [11]. Some other mechanisms, like the GRO (generic receive offload), are also directly supported as a part of the NAPI poll loop.

# Chapter 5

# Implementation of the Network Driver

This section describes the actual implementation details and roadblocks that had to be overcome while creating the kernel driver.

## 5.1 The Driver Initialization

In the Linux operating system, the network interface is represented in the kernel by the `struct net_device`. By interacting with this object and setting its fields, the kernel driver changes the state of the network interface as seen from the user-space.

### 5.1.1 Allocating the Net Device

In order to allocate the net device, the function `alloc_etherdev_mqs` is called. Its parameters are the size of private data, which is a structure that the driver adds to a net device object, which usually holds the metadata needed by the driver to work with the net device. The other two parameters are the number of tx and rx queues to be allocated to the net device. The number of these software queues directly corresponds to the number of hardware queues of the network card. The design of software queues allows for the software traffic shaping by the Linux kernel.

### 5.1.2 Setting up the Defaults of the Net Device Structure

Before registering the net device with the system, the driver needs to set all the desired fields in the `struct net_device`. This is usually done through the helper functions, and for this driver implementation, it includes stopping all software tx queues so that no packets can arrive at the tx path before the driver is ready – `netif_tx_stop_all_queues()` and setting the network link to down, otherwise its status would show up as `UNKNOWN` to the system – `netif_carrier_off()`.

### 5.1.3 Initializing the DMA Controller

The next initialization step is to start up the DMA hardware queues of the network card. This includes allocating and DMA mapping all the memory and passing the references of the control buffers to the DMA controller. Lastly, the reference to the hardware queues

and its control data is stored in the private data of the net device structure so that it can be readily available when working with the net device object.

### 5.1.4 Initializing the Carrier

The driver must enable the carriers on the network card. This is done through the `libnfb` API by opening the `rx_mac` and `tx_mac` components. Furthermore, the driver must periodically check the status of the carrier and ensure that it is reflected by the net device object. This is accomplished in the driver by initializing a kernel work with a 1-second timer which periodically resets itself that checks the state of the carrier through the `libnfb` function `nc_rxmac_get_link()` and then, based on results, either sets the carrier state on – `netif_carrier_on()` or off – `netif_carrier_off()`. This is reflected from the user perspective as the network interface link either being `UP` or `DOWN`.

### 5.1.5 Network Device Kernel Hooks

The main way the kernel interacts with the hardware is by calling the kernel hooks. These are a set of pointers to functions defined in `struct net_device_ops`. They are the kernel-defined API for the network device drivers. This includes, for example, `ndo_open`, which is called on the net device register, or `ndo_start_xmit`, which is called when a packet is to be transmitted. By implementing these functions and then assigning the structure to the net device object, the driver implements its functionality and exposes it to the kernel. If a kernel hook is set to `NULL`, the kernel takes it as a functionality that is not implemented by the driver.

### 5.1.6 Initializing the XDP Functionality

In order to enable the XDP functionality in the driver, the `ndo_bpf`, `ndo_xdp_xmit`, and `ndo_xsk_wakeup` kernel hooks need to be implemented.

- `ndo_bpf` – Responsible for swapping the reference to the XDP program loaded by the user to the driver and for DMA mapping the memory pool of buffers (user-allocated UMEM buffer) when switching to AF_XDP operation and DMA unmapping it when switching back to basic XDP operation.
- `ndo_xdp_xmit` – This function is called when the XDP frame is redirected from another network interface to be transmitted out.
- `ndo_xsk_wakeup` – Used for manual scheduling of sending and receiving packets from the AF_XDP user-space application by means of system calls.

The supported XDP features are set through `xdp_set_features_flag()`, the values are defined in `enum netdev_xdp_act`.

- `NETDEV_XDP_ACT_BASIC` – Basic XDP support.
- `NETDEV_XDP_ACT_REDIRECT` – Basic XDP redirect operation support.
- `NETDEV_XDP_ACT_NDO_XMIT` – Support of transmitting frames redirected from other interfaces.
- `NETDEV_XDP_ACT_XSK_ZEROCOPY` – Support of AF_XDP operation.
- `NETDEV_XDP_ACT_HW_OFFLOAD` – XDP program offloaded onto the network card - not supported by NDK.
- `NETDEV_XDP_ACT_RX_SG` – Support of segmented buffers with basic XDP operation – currently unsupported by the driver.
- `NETDEV_XDP_ACT_NDO_XMIT_SG` – Support of segmented buffers when using redirect to different interface – currently unsupported by the driver.

Lastly, for each queue `struct xdp_rxq_info` has to be created and registered with a call of `xdp_rxq_info_reg()` and the XDP memory model has to be set with a call to `xdp_rxq_info_reg_mem_model()`, the memory models are defined in `enum xdp_mem_type`.

- `MEM_TYPE_PAGE_SHARED` – Split page model – not used in the driver.
- `MEM_TYPE_PAGE_ORDER0` – One page / one packet model – not used in the driver.
- `MEM_TYPE_PAGE_POOL` – Using kernel page pool API to allocate XDP buffers – this is the preferred model for the basic XDP operation and it is used in the driver.
- `MEM_TYPE_XSK_BUFF_POOL` – The AF_XDP operation model – the buffer pool refers to the user-space allocated UMEM buffer.

### 5.1.7 Finishing the Initialization

The initialization is finished by calling the `register_netdev()` function onto the net device object. This calls for the `ndo_open` net device driver hook, which is responsible for starting the receiving and transmitting of packets. From this point onward, the Linux system will show the device as a network interface that is usable from the user-space.

### 5.1.8 Scheduling Peculiarity

Since the NDK FPGA platform does not support interrupts on the RX, for historical reasons – both the NDP API and DPDK technologies use busy polling to handle the scheduling, and there was never a need to support more traditional network operations – the NAPI scheduling has to be handled by the driver itself. For this reason, the driver spawns a kernel thread for each NAPI, which disables software interrupts and calls the `napi_schedule()` function, before enabling them again. The NAPI mechanism could then seem a bit redundant, but it actually makes everything easier, since the usage of NAPI has some guarantees, like that there will only be a single instance of a particular NAPI running. Using XDP requires these guarantees [11]. This leaves the driver in a bit of a

rough spot, because busy polling the RX (and TX in the AF_XDP mode) wastes resources. The overhead of this is minimized in the implementation by using a very short nanosecond sleep, but nevertheless, it is still there.

## 5.2 The Basic XDP Operation

When working in the basic XDP mode (not the AF_XDP zero-copy), the driver allocates its own memory for the buffers. In the driver, the Page Pool API is used for this purpose.

### 5.2.1 Initializing Page Pool

In the driver, each XDP receive queue has its own page pool. The page pool is created by filling the `struct page_pool_params` Listing [5.1] and passing it to the `page_pool_create()` function. The parameters of the page pool include options for automatic DMA mapping and the size of the pool. After creating the page pool, the driver can allocate the page from it by calling `page_pool_dev_alloc_pages()`. While the pages are usually recycled or released later in the network stack, the driver can do so as well by calling `page_pool_put_full_page()`, which will either recycle the page back or release it if its reference count is $> 1$.

```
1 struct page_pool_params ppp = {
2     // Map the pool and synchronize for the device on allocation
3     .flags = PP_FLAG_DMA_MAP | PP_FLAG_DMA_SYNC_DEV,
4     .dev = &nfb->pci->dev, // DMA device
5     .dma_dir = DMA_BIDIRECTIONAL, // DMA direction
6     .max_len = PAGE_SIZE, // Lenght of the DMA mapping
7     .order = 0, // Denotes full page
8     .pool_size = desc_cnt,
9 };
```

Listing 5.1: The starting parameters of page pool

### 5.2.2 Implemeting NAPI

The NAPI is used on the RX as a scheduling mechanism. The parameter `budget` corresponds to the number of packets that are to be processed in one go. The NAPI poll loop ends with a call to `napi_complete_done()` Listing [5.2 R22] with the amount of work done by the loop. The kernel then either reschedules the NAPI again or returns and enables interrupts.

```
1  int nfb_xctrl_napi_poll_pp(struct napi_struct *napi, int budget)
2  {
3      ...
4      // Fill the card with buffers and receive budget of packets
5      received = nfb_xctrl_rx_pp(ctrl, xdp, budget);
6      for (i = 0; i < received; i++) {
7          // Run XDP and handle results
8          nfb_xctrl_handle_pp(ethdev->prog, xdp[i], rxq);
9      }
10     // Flush DMA counters
11     nc_ndp_ctrl_sp_flush(&ctrl->c);
12     ...
13     // Flush XDP redirect maps
14     xdp_do_flush();
15     ...
16     // Work not done -> reschedule if budget remains
17     if (received == budget)
18         return budget;
19
20     // Work done -> finish
21
22     napi_complete_done(napi, received);
23     return received;
24 }
```

Listing 5.2: The NAPI poll loop

### 5.2.3 RX Path

On the RX path, the packet processing is controlled by the NAPI. In the NAPI poll loop, the driver has to fill the NDK DMA ring Listing [5.3 R7-10] with the pages from the page pool. Check if any packets have arrived on the network card, and if so, update the DMA counters to claim the buffers Listing [5.3 R12-14]. The streaming memory mappings has to be synchronized for the CPU access Listing [5.3 R22-25], the packets are now ready for processing. The driver then has to run the XDP program (if present) for each buffer and handle the result. Under normal circumstances, the packets are passed to the kernel network stack. The kernel itself correctly handles all the memory through the use of the Page Pool API.

```
1  static inline u16
2  nfb_xctrl_rx_pp(struct xctrl *ctrl, struct xdp_buff **buffs, u16 nb_pkts)
3  {
4      struct nc_ndp_hdr *hdrs = ctrl->rx.hdr_buffer_cpu;
5      struct nc_ndp_hdr *hdr;
6      ...
7      // Fill the card with empty buffers
8      while (nfb_xctrl_rx_fill_pp(ctrl))
9          ;
10     nc_ndp_ctrl_sdp_flush(&ctrl->c);
11
12     // Update DMA counters
13     nc_ndp_ctrl_hhp_update(&ctrl->c);
14     nb_rx = (ctrl->c.hhp - shp) & mhp;
15     if (nb_pkts < nb_rx)
16         nb_rx = nb_pkts;
17
18     // Ready packets for processing
19     for (i = 0; i < nb_rx; ++i) {
20         hdr = &hdrs[shp];
21         buffs[i] = ctrl->rx.pp.xdp_ring[shp];
22         dma_sync_single_for_cpu(ctrl->dma_dev,
23             page_pool_get_dma_addr(virt_to_page(buffs[i]->data_hard_start)),
24             PAGE_SIZE,
25             DMA_BIDIRECTIONAL);
26         buffs[i]->data_end = buffs[i]->data + hdr->frame_len;
27         ...
28     }
29     ...
30     return nb_rx;
31 }
```

Listing 5.3: The receive function

### 5.2.4 TX Path

TX path is handled through the `ndo_start_xmit` Listing [5.4] kernel hook and consists of collecting the buffers that were already sent out by the network Listing [5.4 R11-12], DMA mapping the packet buffer Listing [5.4 R15], enqueuing it to onto the DMA TX ring Listing [5.4 R32], and saving the metadata of the buffer so that the buffer can be collected after the packet has been sent out Listing [5.4 R28-31]. Finally, the counters are flushed, packets now belong to the card. Listing [5.4 R34]

```
1  netdev_tx_t
2  nfb_xctrl_start_xmit(struct sk_buff *skb, struct net_device *netdev)
3  {
4      ...
5      struct xctrl *ctrl = txq->ctrl; // dma controller object
6      ...
7      // Unlike the RX path which is under NAPI protection
8      // the TX has to be synchronized manually
9      spin_lock(&ctrl->tx.tx_lock);
10     {
11         nc_ndp_ctrl_hdp_update(&ctrl->c); // Update DMA counters
12         nfb_xctrl_tx_free_buffers(ctrl); // Collect the sent buffers
13         ...
14         // DMA map the buffer
15         dma = dma_map_single(ctrl->dma_dev, skb->data, len, DMA_TO_DEVICE);
16         ...
17         // Update the adress if needed
18         last_upper_addr = ctrl->c.last_upper_addr;
19         if (unlikely(NDP_CTRL_DESC_UPPER_ADDR(dma) != last_upper_addr)) {
20             last_upper_addr = NDP_CTRL_DESC_UPPER_ADDR(dma);
21             ctrl->c.last_upper_addr = last_upper_addr;
22             descs[sdp] = nc_ndp_tx_desc0(dma);
23             ctrl->tx.buffers[sdp].type = NFB_XCTRL_BUFF_DESC_TYPE0;
24             ...
25         }
26         // Save the buffer metadata so that it can be collected
27         // after the packet has been sent out
28         ctrl->tx.buffers[sdp].type = NFB_XCTRL_BUFF_SKB;
29         ctrl->tx.buffers[sdp].skb = skb;
30         ctrl->tx.buffers[sdp].dma = dma;
31         ctrl->tx.buffers[sdp].len = len;
32         descs[sdp] = nc_ndp_tx_desc2(dma, len, 0, 0); // Enqueue the buffer
33         ...
34         nc_ndp_ctrl_sdp_flush(&ctrl->c); // Flush the DMA
35     }
36     spin_unlock(&ctrl->tx.tx_lock);
37     return NETDEV_TX_OK;
38 }
```

Listing 5.4: The ndo_start_xmit kernel hook

## 5.3 The AF_XDP Operation

When the driver switches the queue pair to the AF_XDP operation, it has to stop the TX software queue, disable the RX NAPI, and reconstruct the DMA queues. In this mode, the memory is allocated by the user-space. UMEM, the user allocated buffer, is referred to as the XSK buffer pool. The driver interacts with the AF_XDP through the kernel API functions.

### 5.3.1 Mapping the Buffer Pool

When the user-space application registers the UMEM buffer with the AF_XDP socket, the `ndo_bpf` kernel hook is called Listing [5.5], and the buffer is DMA mapped Listing [5.6 R10]. Since the whole XDP memory model is switched, the DMA queues have to be reconstructed Listing [5.6 R14-16].

```
1  int nfb_xdp(struct net_device *dev, struct netdev_bpf *xdp)
2  {
3      int ret;
4      switch (xdp->command) {
5      ...
6      case XDP_SETUP_XSK_POOL:
7          if (xdp->xsk.pool) {
8              ret = nfb_setup_xsk_pool(dev, xdp->xsk.pool, xdp->xsk.queue_id);
9          }
10     ...
11     }
12 }
```

Listing 5.5: The ndo_bpf kernel hook

```
1  static int
2  nfb_setup_xsk_pool(struct net_device *dev,
3      struct xsk_buff_pool *pool,
4      u16 qid)
5  {
6      struct nfb_ethdev *ethdev = netdev_priv(dev);
7      struct nfb_xdp_channel *channel = &ethdev->channels[qid];
8      ...
9      // DMA map the pool
10     xsk_pool_dma_map(pool, &ethdev->nfb->pci->dev,
11         DMA_ATTR_SKIP_CPU_SYNC)));
12     ...
13     // Remake the DMA queues
14     channel_stop(channel);
15     set_bit(NFB_STATUS_IS_XSK, &channel->status);
16     channel->pool = pool;
17     ...
18 }
```

Listing 5.6: Setting up the XSK buffer pool

### 5.3.2 RX Path

On the RX side, the `xsk_buff_alloc_batch()` Listing [5.8 R5] serves for allocating the empty buffers for filling into the card from the buffer pool Listing [5.7 R6-8]. To make the packets ready to be received, the xsk helper functions can be used to set the buffer size and synchronize the DMA streaming mapping for the CPU Listing [5.7 R17-19], after this packets are ready to be processed. In this mode, the intended way is to call the `xdp_do_redirect()` function and redirect the packet to the user-space. This requires the redirecting XDP program loaded onto the network interface and the reference to the XSK socket placed to the eBPF map. If these requirements are not met, the kernel driver must fall back to slow allocating and copying from the user-space allocated buffers. NAPI is used for scheduling and synchronization.

```
1  static inline u16
2  nfb_xctrl_rx_xsk(struct xctrl *ctrl, struct xdp_buff **buffs, u16 nb_pkts)
3  {
4      ...
5      // Fill the card with empty buffers
6      while (nfb_xctrl_rx_fill_xsk(ctrl))
7          ;
8      nc_ndp_ctrl_sdp_flush(&ctrl->c);
9      // Update the DMA counters
10     nc_ndp_ctrl_hhp_update(&ctrl->c);
11     ...
12     // Ready packets for processing
13     for (i = 0; i < nb_rx; ++i) {
14         buff = ctrl->rx.xsk.xdp_ring[shp];
15         hdr = &hdrs[shp];
16         // Set the actual data size after receive
17         xsk_buff_set_size(buff, hdr->frame_len);
18         // Sync the data buffer
19         xsk_buff_dma_sync_for_cpu(buff, ctrl->rx.xsk.pool);
20         ...
21     }
22     return nb_rx;
23 }
```

Listing 5.7: Receive function

```
1  static inline int nfb_xctrl_rx_fill_xsk(struct xctrl *ctrl)
2  {
3      // Allocate the XSK buffers
4      frame_len = xsk_pool_get_rx_frame_size(pool);
5      real_count = xsk_buff_alloc_batch(pool, buffs, batch_size);
6      for (i = 0; i < real_count; i++) {
7          dma = xsk_buff_xdp_get_dma(buffs[I]);
8          ...
9      }
10     ...
11 }
```

Listing 5.8: Filling the card with empty buffers

### 5.3.3 TX Path

Unlike with the basic XDP mode where the TX is handled just in the `ndo_start_xmit()` kernel hook Listing [5.4], with the AF_XDP, the intended way to send packets is through the user-space application with XDP socket. This means that the driver needs to have a separate loop for polling the XDP buffers from the user-space, this is accomplished through calling the `xsk_tx_peek_release_desc_batch()` Listing [5.9 R17] kernel API function. After getting the packets from the AF_XDP, the driver has to get the virtual and DMA addresses Listing [5.9 R19-20] of the data from the buffers so that it can enqueue them to a DMA ring, useful xsk functions exist for this purpose. Another responsibility of the TX loop is competition of the send out buffers Listing [5.9 R10-14]. Marking the buffers as complete returns them to the user-space application. Just like on the RX, the scheduling is handled by NAPI, but this time, the spinlock must also be used for the synchronization Listing [5.9 R7] since nothing prevents any other user-space application from sending the packets to the network interface.

```
1  int nfb_xctrl_napi_poll_tx_xsk(struct napi_struct *napi, int budget)
2  {
3      ...
4      struct xctrl *ctrl = txq->ctrl; \\ the DMA ctrl object
5      ...
6      // TX needs to be synchronized by lock
7      spin_lock(&ctrl->tx.tx_lock);
8      {
9          // Update the DMA counters
10         nc_ndp_ctrl_hdp_update(&ctrl->c);
11         // Free the completed buffers
12         nfb_xctrl_tx_free_buffers(ctrl);
13         // Return the completed XSK buffers to the user-space
14         xsk_tx_completed(pool, ctrl->tx.completed_xsk_tx);
15         ...
16         // Peek for buffers filled by the user-space
17         ready = xsk_tx_peek_release_desc_batch(pool, budget);
18         for (i = 0; i < ready; i++) {
19             data = xsk_buff_raw_get_data(pool, buffs[i].addr);
20             dma = xsk_buff_raw_get_dma(pool, buffs[i].addr);
21             len = buffs[i].len;
22             // Enqueue the buffers to a DMA queue
23             ...
24         }
25         // Flush the DMA counters
26         nc_ndp_ctrl_sdp_flush(&ctrl->c);
27     }
28     spin_unlock(&ctrl->tx.tx_lock);
29
30     // Work not done -> reschedule if budget remains
31     if (i == budget)
32         return budget;
33
34     // Work done -> finish
35     napi_complete_done(napi, i);
36     return i;
37 }
```

Listing 5.9: The TX NAPI poll loop

# Chapter 6

# The AF_XDP User-Space Applications

This chapter introduces the basic idea of AF_XDP application that uses the user-space `libxdp` program in tandem with the kernel-space `libbpf` XDP program loaded onto the network interface to receive packets through AF_XDP socket. Then it describes the actual implementation in the NDK FPGA user-space tools.

## 6.1 Introduction to libxdp

Since the AF_XDP technology is implemented in the Linux kernel, you do not strictly need any libraries to write the AF_XDP program, but doing so would require much unnecessary work. `libxdp` is a way to address this issue. `Libxdp` is a library for attaching XDP programs and using AF_XDP sockets. Specifically, for the XDP side of things, it provides functions to open a compiled XDP program and attach it and detach it from the interface, and functions to manipulate eBPF maps – for passing reference to AF_XDP sockets. AF_XDP part of `libxdp` simplifies the setup of the UMEM memory space, the four buffer rings, and the opening of the AF_XDP socket. Lastly, `libxdp` contains functions to control packet flow by filling the producer rings and polling the consumer rings. Another feature of `libxdp` is loading of the default XDP program, making basic AF_XDP functionality available without the need to provide a custom XDP program for redirecting packets to the user-space [12].

### 6.1.1 The Slow AF_XDP Fallback

When using the AF_XDP sockets with a custom XDP program, it is very important to choose what will happen to the packets that will not be redirected to the user-space. This can happen, for example, when the XDP program cannot find a reference to the AF_XDP socket because no reference was placed in the `BPF_MAP_TYPE_XSKMAP` by the user-space application. If the packets are not dropped, then the driver has to use a very slow fall back option. The reason for this is that since the memory on which the packet resides is owned by the user-space AF_XDP application, when the packet should be, for example, passed to the network stack, the driver has to allocate the memory for it and copy out all the data to this new memory. For this reason, only the redirect to the user-space action and the drop action should be used by the loaded XDP program. If the developer wants to achieve the equivalent of other XDP actions, they should do so by going through the user-space. For example, to retransmit a packet, one should receive it from the AF_XDP RX ring and send

it out again to the AF_XDP TX ring, instead of changing the XDP program to return a `XDP_TX` action, which would trigger a memory allocation and copying. This issue should not be encountered when using the default XDP program loaded with the `libxdp`, since the `libxdp` should take care of placing the socket reference to the `BPF_MAP_TYPE_XSKMAP` automatically.

## 6.2 Structure of a libxdp AF_XDP Program

This whole section is inspired by [12].

### 6.2.1 Attaching the XDP Program

AF_XDP user-space program has to work in tandem with the XDP program in the kernel. Since the most bare-bones XDP program for the AF_XDP operation just calls the `bpf_redirect_map()` function internally, which checks if there is a reference to the socket placed in the eBPF map of type `BPF_MAP_TYPE_XSKMAP` and if so redirects packet to user-space, `libxdp` already provides this program and the user-space application does not have to take care of this step. If more advanced functionality is required, the application has to pass `XSK_LIBBPF_FLAGS__INHIBIT_PROG_LOAD` flag when creating the AF_XDP socket and use the `libxdp` `xdp_program__open_file()` and `xdp_program__attach` functions.

### 6.2.2 Setting up UMEM with FILL and COMPLETION Rings

Setting up the memory space for the packets is done in three steps:

- Allocating the memory for the packets – for the default mode in which the UMEM is made up of chunks of the size of one memory page, the `posix_memalign()` function can be used, to achieve having exactly one packet per memory page.
- Filling out the UMEM parameters into the `struct xsk_umem_config` – These are the size of the COMPLETION ring, the size of the FILL ring, UMEM flags, the size of one frame inside UMEM, and the size of headroom for metadata
- Calling the `xsk_umem__create()` function – This function takes a reference to the allocated memory for packets and a reference to the `struct xsk_umem_config` and in return fills out `struct xsk_umem` representing the UMEM memory, and also a `struct xsk_ring_prod` and `struct xsk_ring_cons`, which represent the FILL and COMPLETION ring respectively. After the successful return of this function, the UMEM, the FILL ring, and the COMPLETION ring are created.

### 6.2.3 Creating AF_XDP Socket with RX and TX Rings

AF_XDP socket corresponds to a single TX/RX channel on the network card. Creating the socket is done in two steps:

- Filling out the `struct xsk_socket_config` structure with parameters – These are flags, including the `XSK_LIBBPF_FLAGS__INHIBIT_PROG_LOAD` flag for loading custom XDP programs, the size of the RX ring, and the size of the TX ring.
- Creating the socket with `xsk_socket__create()` function – This function takes a reference to the `struct xsk_umem`, a reference to the `struct xsk_socket_config`, and in return fills out `struct xsk_socket` which represents the AF_XDP socket and a `struct xsk_ring_prod` and `struct xsk_ring_cons` representing the TX ring and the RX ring respectively. After the successful return of this function, UMEM and all rings are set up, AF_XDP socket is opened, and the application can receive and send packets.

### 6.2.4 Mechanism for Counting Addresses

The application manipulates packets by enqueing them onto and collecting them from the buffer rings descriptors. The packets are represented by the address of the UMEM frame on which they are located and their length. The addresses in this case are multiples of the frame size with the first address being zero. The `libxdp` does not take care of handling the UMEM addresses. The application itself must ensure that the address of a UMEM frame is not used multiple times and is correct. This requires a simple allocation/deallocation mechanism to be implemented by the application. For example, a simple stack with all the addresses into a UMEM.

### 6.2.5 RX Path

On the RX path, the application needs to first fill the RX, which means giving empty UMEM frames to the network card through the FILL ring so that the card can fill the memory with packet data. This is accomplished by calling the `xsk_ring_prod__reserve()` function to reserve space on the FILL ring, filling the descriptors with addresses into the UMEM through the `xsk_ring_prod__fill_addr()` function, and finishing the process of submitting by calling the `xsk_ring_prod__submit()` function. Next, the application can check the RX ring for any incoming packets by calling the `xsk_ring_cons__peek()` function. If any packets are received, the application can process them by calling the `xsk_ring_cons__rx_desc()` function to get the RX ring descriptor and then use the function `xsk_umem__get_data()` to get the pointer to the packet data.

### 6.2.6 TX Path

On the TX path, the application has to enqueue the packets onto the TX ring to be sent out by the network card, this is accomplished by calling the `xsk_ring_prod__reserve()` function to reserve the space on the TX ring and then filling out the descriptors with the help of the `xsk_ring_prod__tx_desc()` and `xsk_umem__get_data()` function, the TX is finished by calling the `xsk_ring_prod__submit()` function. After this, the application has to collect the completed (sent out) addresses so that they can be used again. This is done by calling the `xsk_ring_cons__release()` function after which the free addresses are accessible through the `xsk_ring_cons__comp_addr()` function.

## 6.3 Implementation to the NKD FPGA Platform User-Space Tools

The actual implementation of AF_XDP into NDK FPGA platform tools is done as a part of its `ndp-tool` application. This application is one of the testing and benchmarking applications for the NKD FPGA framework and has a modular design. Although there is only a single binary, each mode is available as its own executable pseudo-application through a symlink. The available modes are:

- `ndp-generate` – Generates packet at a given speed.
- `ndp-read` – Receives packets.
- `ndp-transmit` – Replays packets from a file at a given speed.
- `ndp-receive` – Receives packets into a file.
- `ndp-loopback` – Creates a software loopback, received packets are sent out.

The application uses the NDP API for its function. DPDK version of the application also exists in the NDK FPGA software, which adds new modes (`ndp-dpdk-generate` and so on) but is not compiled by default, because of its reliance on the third-party DPDK library. The implementation of the XDP modes is done in the same way as the DPDK implementation. So far only `ndp-xdp-read` and `ndp-xdp-generate` are available. This allows for basic benchmarking of the NDK FPGA's own NDP API, the DPDK implementation, and now the XDP implementation against each other.

### 6.3.1 Common Parameters

All modes of the `ndp-tool` application share some parameters, the most important for the XDP implementation are these:

- `-d path Path to device [default: /dev/nfb0]` – Important for choosing the testing device.
- `-i indexes Queues numbers to use - "0-5,7" [default: all]` – Important because in order to correctly map these indices to the XDP queues made available by the kernel driver, the driver has to make this information available in some way. The `sysfs` kernel interface serves this purpose.

## 6.4 Mode ndp-xdp-read of ndp-tool Application

The `ndp-xdp-read` application opens up AF_XDP sockets based on the parameters passed and tries to receive as many packets as possible. It is used to measure the speed of the receive path. The application uses the default XDP redirecting program from `libxdp`, meaning that no XDP program has to be shipped together with the application, and the program and the socket are opened in native mode, leveraging the kernel driver support. To correctly map the NDK FPGA queues to the XDP interfaces created by the kernel driver, the application reads `sysfs` files created by the kernel driver. The application shows statistics in the terminal window with the help of `libncurses` library in real time. The application does not have any new parameters.

## 6.5 Mode ndp-xdp-generate of ndp-tool Application

The `ndp-xdp-generate` application opens up AF_XDP sockets based on the parameters passed and tries to send out as many packets of the particular length as possible. The length is passed to the application as a new parameter. The packet data can either be memory garbage or can be set to zero. Since the application only sends packets and does not receive any, the XDP redirecting program is not necessary for the operation. The AF_XDP socket is opened in the default native mode, leveraging kernel driver support. The `XSK_LIBBPF_FLAGS__INHIBIT_PROG_LOAD` flag is passed on the AF_XDP socket creation. To correctly map the NDK FPGA queues to the XDP interfaces created by the kernel driver, the application reads `sysfs` files created by the driver. The application shows statistics in the terminal window with the help of `libncurses` library in real time. The application introduces three new parameters.

### 6.5.1 Parameters

- `-s size Packet size - list or random from range` – The size of generated packets, this parameter is required.
- `-C Clear packet data before send` – Whether to set the packet data to zero, option parameter.
- `--speed Mbps Replay packets at a given speed` – Caps the speed at which packets are generated.

# Chapter 7

# Testing

This chapter talks about testing of the XDP driver implementation by loading simple XDP programs and benchmarking the driver using the `ndp-tool` application.

## 7.1 Testing of the Default XDP Operation

The default XDP operation was tested using a network card with two physical network interfaces that were connected together with an optical cable to perform a hardware loopback. The open-source program `tcpreplay`[1] was used to generate traffic on the interface from a pcap file. The open-source program `tcpdump`[2] was used to dump the traffic on the ports. These programs were selected because they are known, tested, and well-documented open-source programs, and their results can be safely compared with the expected values. The elementary XDP programs for each of the XDP return codes were loaded onto the interfaces, and their effect was checked against the expected results. The `xdp-loader` utility tool, which comes as part of `xdp-tools` package[3] together with `libxdp`, was used for managing the XDP programs.

### 7.1.1 Tests Performed

- `XDP_DPOP` – After loading the program, no packets were dumped by `tcpdump`.
- `XDP_PASS` – the packets were passed to the network stack and were dumped by `tcpdump`.
- `XDP_TX` – the packet was transmitted from the same interface it arrived on, `tcpdump` on the interface the XDP program was loaded on, dumped no packets, and the `tcpdump` that was run on the interface the packets were retransmited to made them correctly show up.
- `XDP_REDIRECT` – the packet was redirected to the same interface it arrived on, mimicking the `XDP_TX` action, just like before, `tcpdump` on the interface the XDP program was loaded on, dumped no packets, and the `tcpdump` that was run on the interface the packets were retransmited to made them correctly show up.

[1] https://tcpreplay.appneta.com/
[2] https://www.tcpdump.org/
[3] https://github.com/xdp-project/xdp-tools

### 7.1.2 Conlcusion

All tests were passed and the driver works as expected. All XDP actions were performed correctly and no packets were lost.

## 7.2 AF_XDP Benchmarks and Comparisons to NDP and DPDK

Benchmarking is performed by running the `ndp-tool` application, which is a part of the NDK FPGA software and was extended to be AF_XDP capable. The `ndp-tool` application is an application that can be run under many different modes, from now on, whenever there is a reference in this section to an application that starts with „`ndp-*`“ it refers to that specific mode of the `ndp-tool`. The tested card had two physical ports capable of running at 100 Gbit speeds, the ports were connected together via optical cable so that a physical loopback was achieved. The tests are performed by measuring the performance of a single hardware DMA queue, running over several seconds with a set packet size. The XDP results are compared against the NDK FPGA custom NDK API and DPDK.

### 7.2.1 Technologies Being Compared

Right now, the `ndp-tool` application supports sending and receiving packets via NDP, DPDK, and XDP/AF_XDP.
The NDP is an API made for the NDK FPGA platform. Its main advantages are being very easy to use and being fully integrated with the NDK FPGA platform, as the API offers functions to control the state of the card, like toggling the ethernet interface. There is also a Python version of the API.
DPDK (Data Plane Development Kit) is a platform for developing network applications. NDK FPGA offers a special user-space driver, which can be used to develop and run DPDK applications. The NDK FPGA supports two modes of operation, one is the NDP DPDK mode, which makes the DPDK run NDP functions under the hood. The other is the native DPDK mode, which uses its own implementation. Some of the advantages of the DPDK are the easy to use support of hugepages – instead of normal memory pages, which are usually 4096 bytes, the DPDK, as a default, supports hugepages, which are usually 1 GB. The DPDK is also well known and widely adopted, making it a good and stable platform for development.
XDP (eXpress Data Path) is in the context of these two technologies unique, because it does not use a special driver to go around the kernel network stack (which is the limiting factor for fast packet processing), but instead works with the network stack, it uses a fast path which is build into the kernel to redirect packets to user-space based on the result of the XDP eBPF program. The main advantages of this are still being able to use the network stack, which is not possible with the other two solutions, while still being comparable in terms of speed, easy portability, the program will run under any new Linux distribution and having no third party dependencies, since the XDP is built into the Linux, with the exception of maybe `libxdp`, which makes programming AF_XDP much easier, but this library is very small and open-source.

| Tech / pkt size | 64 | 128 | 256 | 512 | 1024 | 1496 |
|---|---|---|---|---|---|---|
| NDP L1 [Mbps] | 7374 | 12653 | 23558 | 43854 | 71199 | 86072 |
| NDP L2 [Mbps] | 5671 | 10988 | 21876 | 42218 | 69840 | 84943 |
| NDP [Mpps] | 10,426 | 10,406 | 10,517 | 10,227 | 8,492 | 7,06 |
| NDP DPDK L1 [Mbps] | 7448 | 13081 | 23697 | 41203 | 64984 | 91617 |
| NDP DPDK L2 [Mbps] | 5755 | 11360 | 22004 | 39665 | 63744 | 90412 |
| NDP DPDK [Mpps] | 10,58 | 10,758 | 10,579 | 9,609 | 7,751 | 7,534 |
| DPDK L1 [Mbps] | 9775 | 16095 | 31491 | 58074 | 94440 | 91358 |
| DPDK L2 [Mbps] | 7553 | 13977 | 29242 | 55907 | 92637 | 90156 |
| DPDK [Mpps] | 13,885 | 13,236 | 14,059 | 13,544 | 11.259 | 7,512 |
| XDP L1 [Mbps] | 9695 | 16686 | 30344 | 58145 | 99373 | 99139 |
| XDP L2 [Mbps] | 7491 | 14491 | 28176 | 55975 | 97476 | 97835 |
| XDP [Mpps] | 13,772 | 13,723 | 13,547 | 13,56 | 11,853 | 8,153 |

Table 7.1: The measured values as they were collected from the ndp-tool. Single DMA Queue TX, 100 Gbps port, nfb-200G2QL NIC.

### 7.2.2 Testing TX Performance

The TX performance is tested by running the `ndp-generate`, the `ndp-dpdk-generate`, and the `ndp-xdp-generate` applications. The application generates packets of a given size as quickly as possible and prints the results on the terminal. The tested packet sizes are 64B, 128B, 256B, 512B, 1024B, and 1496B. The tested technologies are NDP, DPDK in its NDP mode, DPDK in its native mode, and AF_XDP. Performance is shown as the amount of data transmitted, and results are in megabits per second and megapackets per second. All results are shown in the Table 7.1. The L1 and L2 refer to transmission speed, including overhead, at the physical layer and link layer, respectively. For visualization, the plot of L2 speeds is chosen Figure 7.1.

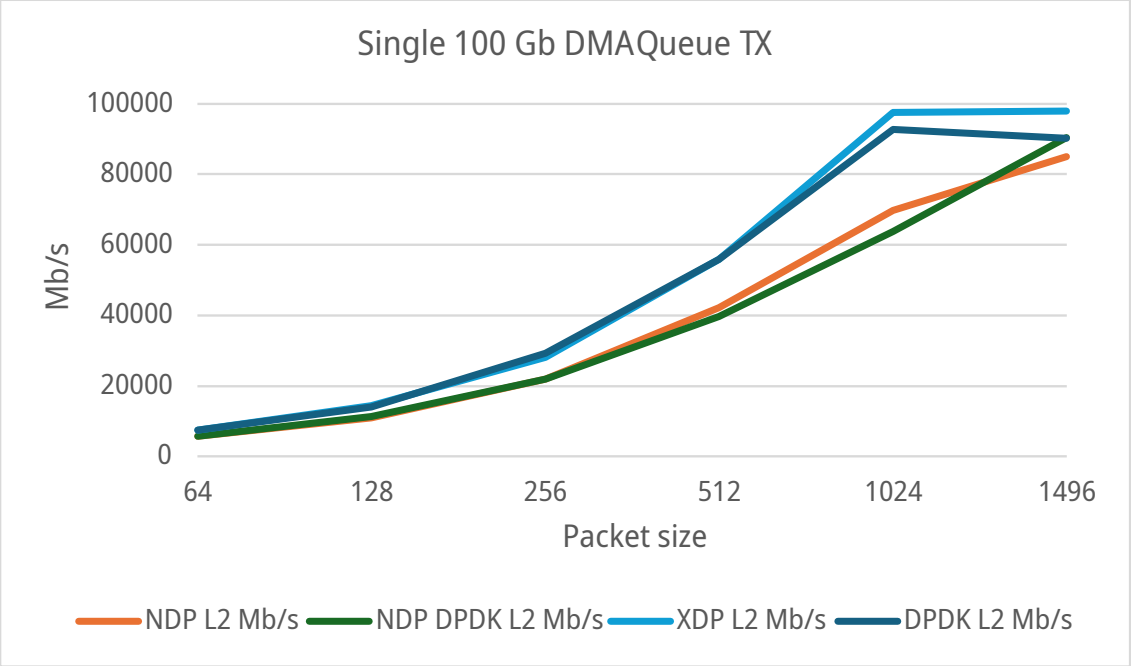


Figure 7.1: Comparison of different technologies, transmitting packets on single hardware DMA queue.

.

### 7.2.3 Conlcusion

The testing showed that on TX, the AF_XDP hits the 100GB port cap sooner than the NDP DPDK and the NDP API. The AF_XDP went head to head with the native implementation of DPDK, the only differences are seen with packets above 1024B, where DPDK seems to have hit some sort of performance cap. This is a very promising result, but we have to keep in mind that the tests were performed under a controlled environment on a very basic workload without any heavy tweaking of the parameters.

## 7.3 Testing RX Performance

The RX performance is tested by setting the card firmware to combine all traffic into a single hardware DMA queue. The tested card had two physical ports capable of running at 100Gb/s, the ports were connected together via optical cable so that a physical loopback was achieved. On one port, the `ndp-read`, the `ndp-dpdk-read` and the `ndp-xdp-read` applications are used to receive packets as fast as possible. The traffic on the other port is generated through a `ndp-generate`, by running the application in a multiqueue mode we ensure that the sending port is maxed out and delivers full 100Gb/s. The tested packet sizes are 64B, 128B, 256B, 512B, 1024B, and 1496B. The tested technologies are NDP, DPDK in its NDP mode, DPDK in its native mode, and AF_XDP. Performance is shown as the amount of data received, and results are in megabits per second and megapackets per second. All results are shown in the Table 7.2. The L1 and L2 refer to transmission speed, including overhead, at the physical layer and link layer, respectively. For visualization, the plot of L2 speeds is chosen Figure 7.2.

| Tech / pkt size | 64 | 128 | 256 | 512 | 1024 | 1496 |
|---|---|---|---|---|---|---|
| NDP L1 [Mbps] | 7522 | 12755 | 22925 | 42803 | 75922 | 92012 |
| NDP L2 [Mbps] | 5812 | 11077 | 21287 | 41206 | 74473 | 90802 |
| NDP [Mpps] | 10,685 | 10,49 | 10,235 | 9,982 | 9,056 | 7,567 |
| NDP DPDK L1 [Mbps] | 7095 | 11347 | 19789 | 28979 | 46477 | 62128 |
| NDP DPDK L2 [Mbps] | 5482 | 9854 | 18375 | 27898 | 45590 | 61310 |
| NDP DPDK [Mpps] | 10,078 | 9,331 | 8,835 | 6,758 | 5,544 | 5,109 |
| DPDK L1 [Mbps] | 14412 | 24435 | 45084 | 84368 | 100000 | 100000 |
| DPDK L2 [Mbps] | 11136 | 21220 | 41864 | 81220 | 98092 | 98684 |
| DPDK [Mpps] | 20,472 | 20,095 | 20,127 | 19,675 | 11,928 | 8,224 |
| XDP L1 [Mbps] | 6298 | 11317 | 20124 | 41752 | 80348 | 96582 |
| XDP L2 [Mbps] | 4867 | 9828 | 18687 | 40194 | 78814 | 95311 |
| XDP [Mpps] | 8,947 | 9,307 | 8,984 | 9,737 | 9,584 | 7,943 |

Table 7.2: The measured values as they were collected from the ndp-tool. Single DMA Queue RX, 100 Gbps port, nfb-200G2QL NIC.

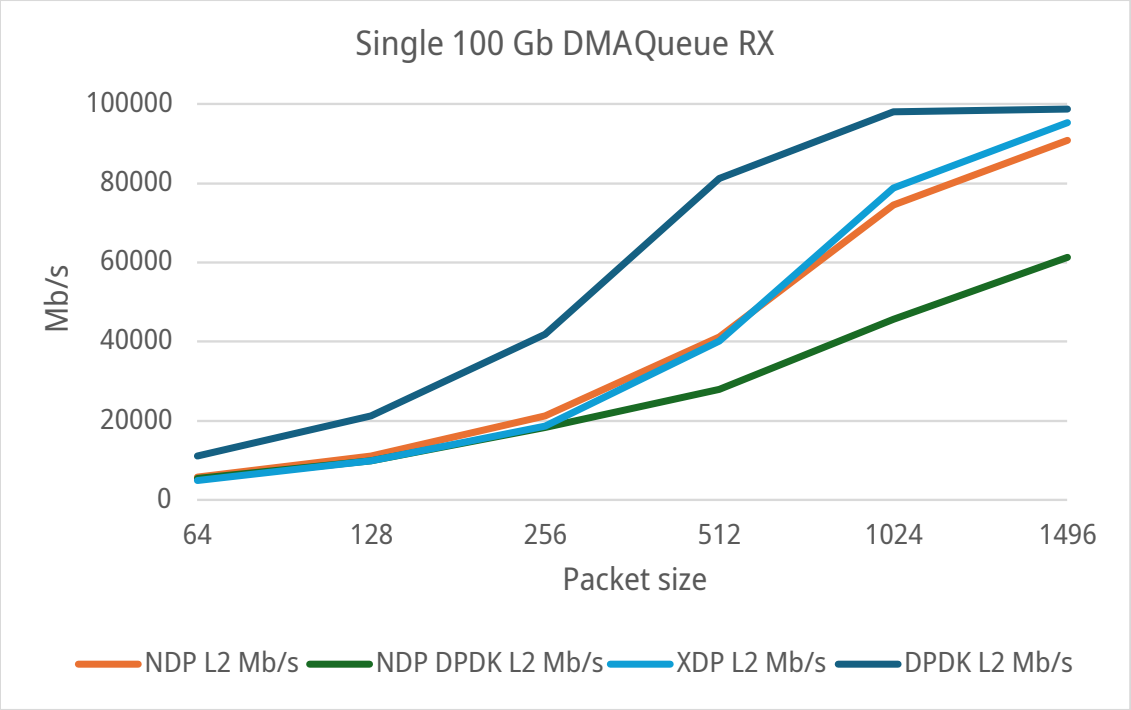


Figure 7.2: Comparison of different technologies, transmitting packets on single hardware DMA queue.

.

### 7.3.1 Conlcusion

The testing showed that on RX, the AF_XDP performs within the margin of error with the NDP API The NDP DPDK fell behind, which is expected due to the copying nature of the technology. The DPDK running in native mode is a clear winner here, this could be due to its usage of hugepages (1GB memory pages) and by virtue of the DPDK implementation

being much more mature than this XDP driver. The AF_XDP application here does not use hugepages, although it is theoretically possible by using `mmap()` with `MAP_HUGETLB`, unlike DPDK, which offers a tool `dpdk-hugepages` written in Python, the AF_XDP would require the user to set up the pages manually and handle everything. This is something, that could be developed as an addition to the NDK FPGA platform in the future. Nevertheless, this is a promising result, which shows the potential of the XDP to at least match the performance of the currently used technologies like NDP, but it shows that to be really comparable to the DPDK, the AF_XDP application would have to be fine tuned and lacks the ease of setup of the DPDK when it comes to hugepage support. Still, we have to keep in mind that the tests were performed under a controlled environment on a very basic workload, and the applications were run without any heavy tweaking of the parameters.

# Chapter 8

# Conclusion and future work

The goal of this thesis was to create an implementation of the XDP-capable network driver that could load XDP programs and leverage the redirect to user-space function to run AF_XDP applications. This goal was accomplished and tested with the basic XDP programs and basic AF_XDP applications for generating and receiving packets. The testing showed promising results and shows XDP as a promising addition to the NDK FPGA framework. There was no effort put into further optimizing the XDP network driver, and this work is not prioritized at this time. The side-effect of implementing the XDP-capable network driver is that when no XDP program is loaded, the driver acts as a usual Linux network driver, which further expands the possibilities of the NDK FPGA framework.

## 8.1 Future Work

This is the work that only became apparent after the implementation was done or was not necessary for the initial implementation.

### 8.1.1 Dynamic Adding and Deleting of the Linux Network Devices

One of the requirements that only became apparent after finishing implementation was to dynamically open and close XDP net devices. The current idea is to use the NDK FPGA software tool `nfb-dma` with parameters to open and close XDP devices with specific hardware DMA queues after the driver has already been loaded. This method would be the same for all other NDK FPGA drivers that add Linux network devices. The method would use `sysfs` interface to signal the driver that the network device should be opened or closed. This is a near-future work and one of the requirements for the driver to be production-ready.

### 8.1.2 Support of Segmented Buffers in the XDP Network Driver

The driver can signal support for this feature by using the `NETDEV_XDP_ACT_RX_SG` feature flag. So far, the size of the packet is limited to one memory page. This is fine as long as the user operates within the default MTU (Maximum Transmission Unit) of the NDK FPGA firmware, which is 1500 bytes. This feature has a low priority.

### 8.1.3 Support Transmitting Segmented Buffers Redirected to the Driver

The driver can signal support for this feature by using the `NETDEV_XDP_ACT_NDO_XMIT_SG` feature flag. This feature allows for a very specific use case of a driver with a segmented buffer implementation to retransmit those buffers back. This feature would have to be implemented together with the support of segmented buffers. This feature has a low priority.

### 8.1.4 Optimizations for the Basic Network Driver Operation

While it is entirely possible to use the driver without the XDP program for a normal Linux network driver operation, there was no effort to optimize this use case, as this goes beyond the scope of this thesis. This is mostly visible when trying to transmit a large number of packets, because there is no attempt to batch the packets on the TX, and the DMA counters are flushed for each transmitted packet. On the RX, the GRO offload could be used to further speed up the receive path (see `napi_gro_receive()` function). This feature has a low priority.

### 8.1.5 Expand the ndp-tool with other XDP Modes

So far, only `ndp-xdp-generate` for generating traffic and `ndp-xdp-read` for receiving traffic have been implemented. Equivalents for the remaining `ndp-tool` modes, mainly `ndp-xdp-transmit` for replaying packets from a file, `ndp-xdp-receive` for recording received traffic to a file, and `ndp-xdp-loopback` for software loopback, are still missing. This feature has a low priority.

### 8.1.6 Expand the ndp-tool XDP Modes with Hugepages Support

When using the AF_XDP modes of the `ndp-tool`, the UMEM memory is allocated via the `possix_memalign()` function, and one XDP frame has the size of a full page and is aligned to be exactly at the page boundaries. Theoretically, it is possible to use `mmap()` with `MAP_HUGETLB` to allocate hugepages, but the pages have to be set up beforehand. DPDK uses a similar mechanism and has its own tool `dpdk-hugepages` written in Python for setting up the pages. Perhaps the same functionality could be added to the `ndp-tool` as well. This feature could make the applications perform closer to the DPDK native mode, so this feature should be looked at.

### 8.1.7 Expand the ndp-tool XDP Modes with More Parameters

Right now the AF_XDP modes of `ndp-tool` use default `libxdp` parameters wherever possible. Some useful additions could be parameters to change the number of descriptors, changing the size of UMEM, setting XDP program loading mode (currently the program is loaded in a native mode, which is preferred, but SKB mode could prove useful for testing), possibility to change the AF_XDP socket flags, possibility to load your own XDP program at the start of the application.

### 8.1.8 Rework of Scheduling/Interrupt in the NDK FPGA Platform

As talked about in Scheduling Peculiarity 5.1.8, the fact that the NDK FPGA platform does not support interrupts means that the driver has to busy poll the RX, which creates unnecessary overhead. Adding the interrupt to the NDK FPGA platform could be a solution, but

as a feature that would only be used by the XDP driver, it is not high on the priority list. There is one more option, and that is to make use of the `XDP_USE_NEED_WAKEUP` flag, which makes the AF_XDP application control the polling through system calls, but that again creates overhead and does not solve the problem of non AF_XDP queues busy polling in the background. This feature should be looked at in the near future.

# Bibliography


[1] *BPF and XDP Reference Guide*. Cilium. Available at: https://docs.cilium.io/en/stable/bpf/.

[2] CESNET z.s.p.o.. *The DMA module*. Available at: https://cesnet.github.io/ndk-fpga/devel/ndk_core/doc/dma.html.

[3] CESNET z.s.p.o.. *NDK Architecture*. Available at: https://cesnet.github.io/ndk-fpga/devel/ndk_core/doc/readme.html.

[4] CESNET z.s.p.o.. *NFB framework*. Available at: https://cesnet.github.io/ndk-sw/index.html.

[5] Corbet, J. Accelerating networking with AF_XDP. *LWN.net*, 2018. Available at: https://lwn.net/Articles/750845/.

[6] Høiland Jørgensen, T.; Brouer, J. D.; Borkmann, D.; Fastabend, J.; Herbert, T. et al. The eXpress data path: fast programmable packet processing in the operating system kernel. In:. New York, NY, USA: Association for Computing Machinery, 2018, p. 54–66. CoNEXT '18. ISBN 9781450360807. Available at: https://doi.org/10.1145/3281411.3281443.

[7] Karlsson, M. and Töpel, B. The Path to DPDK Speeds for AF XDP. In: *Linux Plumbers Conference*. 2018. Available at: https://lpc.events/event/2/contributions/99/.

[8] Miller, D. S.; Henderson, R. and Jelinek, J. *Dynamic DMA mapping Guide*. Available at: https://docs.kernel.org/core-api/dma-api-howto.html.

[9] The kernel development community. *NAPI*. Available at: https://docs.kernel.org/networking/napi.html.

[10] The kernel development community. *Page Pool API*. Available at: https://docs.kernel.org/networking/page_pool.html.

[11] The kernel development community. *Redirect*. Available at: https://docs.kernel.org/bpf/redirect.html.

[12] Toke Høiland-Jørgensen and Magnus Karlsson. *Libxdp – library for attaching XDP programs and using AF_XDP sockets*. Available at: https://github.com/xdp-project/xdp-tools/blob/main/lib/libxdp/README.org.