POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master Degree Thesis

Handling QoS with eBPF: rate limiting and packet scheduling approaches in XDP

Supervisors

Prof. Fulvio RISSO

Ing. Federico PAROLA

Candidate

Andrea AURELI

ACADEMIC-YEAR 2022/2023

Summary

Optimizing and speeding up packet forwarding over networks and performing critical network functions is certainly a challenge in modern networking. Although the Linux kernel networking stack provides a rich and consolidated set of capabilities, its use in high traffic applications is not ideal, since its support of a wide set of use cases comes at the cost of performance. The eBPF technology present in the Linux kernel allows attaching verified programs to specific hooks in the kernel, thereby extending the kernel itself. Running these programs enables bypassing certain steps in the Linux networking stack that may be unnecessary in specific situations. The primary emphasis of this work is on using the eBPF hook known as Linux eXpress Data Path (XDP), which introduces a faster data path for packet processing into the Linux kernel. Although XDP has an optimal behavior in packet forwarding, it does not currently offer a mechanism to manage packet queuing and to implement packet scheduling logic, which is fundamental to enforce Quality of Service (QoS). This work examines the existing features of XDP for managing QoS, evaluating the performance of various possible strategies. The thesis also explores the features offered by a recent patch which represents an ongoing attempt to design a programmable packet scheduling extension for XDP. The patch incorporates recently developed schemes for programmable queues, enabling the creation of packet schedulers in eBPF while still benefiting from the speed of XDP. The thesis work investigates advantages and disadvantages between the features provided by the patch and the queueing discipline used by the Linux traffic control system. The patch guarantees a reduced resource overhead for performing the same scheduling algorithms. In addition, we developed a prototype based on eBPF, capable of autonomously managing routing and rate limiting functions. The prototype achieves better performance compared to the vanilla Linux kernel in terms of throughput and resource utilization, while also ensuring excellent scalability.

Acknowledgements

To my family.
To the sublime talent of Arjen Robben.

Table of Contents

List of Figures VII

Acronyms X

1 Introduction 1
- 1.1 Goal of the thesis . . . 1
- 1.2 Tiesse S.p.a. . . . 2

2 Background 3
- 2.1 eBPF(Extended Berkeley Packet Filter) . . . 3
  - 2.1.1 vCPU . . . 3
  - 2.1.2 Safety . . . 4
  - 2.1.3 Maps . . . 5
  - 2.1.4 Helpers . . . 7
  - 2.1.5 Tail & Function Calls . . . 8
  - 2.1.6 Hook-points . . . 8
  - 2.1.7 eXpress Data Path (XDP) . . . 9
  - 2.1.8 Traffic Control (TC) . . . 10
  - 2.1.9 Toolchain . . . 11
- 2.2 Traffic Control in the Linux Kernel . . . 13
- 2.3 Packet Queuing and scheduling in XDP . . . 14
  - 2.3.1 PIFO and Eiffel extension . . . 15
  - 2.3.2 First attempt: Adding a dequeue hook . . . 16
  - 2.3.3 Second attempt: Using BPF Timer . . . 18

3 Prototype Architecture 21
- 3.1 General Architecture . . . 21
  - 3.1.1 Routing Acceleration . . . 21
  - 3.1.2 Rate Limiter . . . 23
  - 3.1.3 Traffic policing and traffic shaping in the prototype . . . 26

4 Prototype Implementation 28
4.1 BPF skeleton . . . 28
4.2 TC-egress program . . . 31
4.3 XDP program . . . 35
4.4 Rate Limiter different implementations . . . 36
4.4.1 Refilling tokens using BPF Timer . . . 40
4.4.2 Refilling tokens using Perf Event . . . 43
4.4.3 Refilling tokens from Userspace . . . 44

5 Prototype Evaluation 46
5.1 Testbed Setup . . . 46
5.2 Benchmarking tools . . . 47
5.2.1 Iperf3 . . . 47
5.2.2 Cisco TRex . . . 47
5.3 Routing tests . . . 48
5.4 Accuracy tests . . . 49
5.4.1 Comparison among the various approaches presented . . . 51
5.5 Scalability test . . . 54
5.6 LS1046A Freeway Board . . . 54

6 Benchmarking XDP scheduling with PIFO map 58
6.1 Baseline impact of the PIFO map . . . 58
6.2 Strict priority . . . 60
6.3 Weighted Fair Queueing . . . 63
6.4 PIFO's expressiveness . . . 69

7 Conclusions and future work 73
7.1 Future developments . . . 74

Bibliography 76

List of Figures

2.1 Scheme of the verification steps 5
2.2 Interaction with eBPF maps 6
2.3 Tail calls between eBPF programs 8
2.4 Comparison between XDP Generic and Native from [5] 10
2.5 Linux Network Stack 11
2.6 eBPF Toolchain 12
2.7 PIFO data structure: packets can be enqueued at a specific priority, but dequeue only occurs from the head. 16

3.1 Flowchart for the routing process 23
3.2 Token Bucket algorithm in traffic policing 25
3.3 Flowchart for implementing traffic shaping 27

4.1 Flowchart of the eBPF skeleton generation 32
4.2 Flowchart of the entire prototype 37
4.3 With token refilling performed by a userspace thread for more precise behavior, global variables were used instead of maps. 45

5.1 Testbed setup 47
5.2 Throughput results varying packet size of one UDP flow 50
5.3 Accuracy test with TCP while varying the rate to be adhered to for the flow 52
5.4 Accuracy test with UDP while varying the rate to be adhered to for the flow 52
5.5 Occupancy of the core to which traffic is redirected varying packet size - UDP flow 53
5.6 Scalability test - measuring packet rate increasing number of UDP flows 55
5.7 Testbed setup used to test LS1046A Freeway Board 56
5.8 Throughput result varying packet size of one UDP flow - LS1046A Freeway Board 57

5.9 Percentage of CPU idle varying packet size of one UDP flow - LS1046A Freeway Board 57

6.1 Comparison between bridge, vanilla XDP Redirect and XDP Scheduling(packet passes through PIFO map) - One UDP flow varying packet size 59
6.2 Comparison in throughput and core utilization between bridge, vanilla XDP Redirect and XDP Scheduling(packet passes through PIFO map) - One TCP flow 60
6.3 Testbed setup used in the test for strict priority: DUT acts as a bottleneck 63
6.4 Comparison in throughput and core utilization between PRIO Qdisc and XDP program - 2 TCP Flows redirected to one core 64
6.5 Comparison in throughput and core utilization between HTB Qdisc and XDP program - 2 TCP Flows redirected to one core 69
6.6 Example of building a hierarchy with PIFO maps. In the parent map, tags are inserted to indicate from which map to perform the next dequeue 71

Acronyms

DUT
: Device Under Test

DPDK
: Data Plane Development Kit

eBPF
: extended Berkeley Packet Filter

HTB
: Hierarchical Token Bucket

IP
: Internet Protocol

JIT
: Just-In-Time

NIC
: Network Interface Card

QoS
: Quality of Service

RSS
: Receive Side Scaling

SKB
: Socket Buffer

TOS

Type of Service

TCP

Transmission Control Protocol

TC

Transmission Control

UDP

User Datagram Protocol

WFQ

Weighted Fair Queueing

XDP

eXpress Data Path

Chapter 1

Introduction

1.1 Goal of the thesis

In recent years, eBPF/XDP technology has demonstrated higher performance compared to the performance of the vanilla Linux kernel, bringing substantial enhancements to the system. The Linux network stack can be a potential bottleneck due to the speed and dynamism required by modern networks. This has led to the rewriting of numerous network functions for specific issues using eBPF to overcome this limitation. Executing such functions at the early stages of the network stack ensures lower overhead, allowing an overall system optimization.
One fundamental challenge in networking is Quality of Service (QoS) management. The Linux kernel provides a traffic control system called queuing discipline (Qdisc), but using this system does not enable bypassing the Linux network stack. This thesis delves into the possibility of replacing the traffic control system with functions injected in the kernel through eBPF. It explores the capabilities that eBPF technology offers today and its impact on the system.

1.2 Tiesse S.p.a.

Tiesse is an italian privately-owned company focused on innovative network devices, advanced network equipment, corporate business routers and IoT enabling appliances. Tiesse is headquartered in Ivrea close to Turin, and has additional offices in Rome, Turin and Avezzano (near L'Aquila).
Over the years, Tiesse has established itself as one of the leading and most dependable brands in the global market for innovative networking products.
Tiesse products are at the cutting-edge technology thanks to continuous R&D investment and close collaboration with both telco and university research laboratories. In fact, the company has been in constant collaboration for years now with the Turin Polytechnic. Their customers are large companies in sectors such as energy, finance, industry and the public and defence domains.
Tiesse is ISO 9001, UNI EN ISO 14001:2015 and UNI EN ISO 14064-1:2019 certified.

Chapter 2

Background

2.1 eBPF(Extended Berkeley Packet Filter)

eBPF is a technology developed within the Linux kernel that enables to execute sandboxed programs within a privileged context such as the operating system kernel [1]. Officially incorporated in the Linux kernel since version 3.18, eBPF can fully leverage its advanced capabilities with a kernel version of 4.x or higher.
The operating system kernel is the ideal environment for implementing functions related to security, observability and networking, which grants it control over the entire system. However, adding a new functionality or modifying an existing one in the kernel is a time-consuming process, as it typically involves recompiling the entire kernel and rebooting the system each time a new Linux kernel module (LKM) is needed. eBPF eliminates this necessity by allowing developers to write code that runs within the Linux kernel space. The advantages and key features of this technology will be detailed in the following sections.

2.1.1 vCPU

eBPF evolved from its predecessor, BPF (also known as cBPF), which was a general-purpose, event-driven virtual CPU integrated into the Linux kernel starting with

version 2.1.75 in 1997. In its early stages, BPF was primarily utilized as a packet filter within the packet capture tool tcpdump. The eBPF virtual CPU, on the other hand, is composed of eleven 64-bit registers, each of which has 32 bit subregisters. Additionally, it includes a program counter and a 512 byte large eBPF stack space. These registers are denoted as r0 through r10, with r10 being designated as read-only, containing the frame pointer address necessary for accessing the eBPF stack space. The remaining registers, r0 through r9, are considered general-purpose and can be both read from and written to.

2.1.2 Safety

eBPF programs can be dynamically created and injected in the kernel at run-time. Since the program runs in kernel space and not in userspace, dynamic injection of malicious code could lead to compromise of the machine, which is not acceptable. Before injecting the code, the operating system must have no security risk.
Before an eBPF program is attached to the requested hook within the Linux kernel, it passes through the verification step. This step ensures the safety of the eBPF program. It checks various conditions, such as the program does not cause system crashes or any other harm. It must check that the program size is bounded. Then for each possible execution path it verifies that there is a maximum number of instructions (initially set to 4096 and has been raised to above 1 million in kernel 5.1). The program must not include unbounded loops: initially also small loops were rejected then the verifier has been improved. The loops are unrolled to be executed.
The verification step leads to the conclusion that we cannot write any kind of program, so eBPF does not implement a Turing complete machine. The steps undertaken by an eBPF program are shown in the figure 2.1.

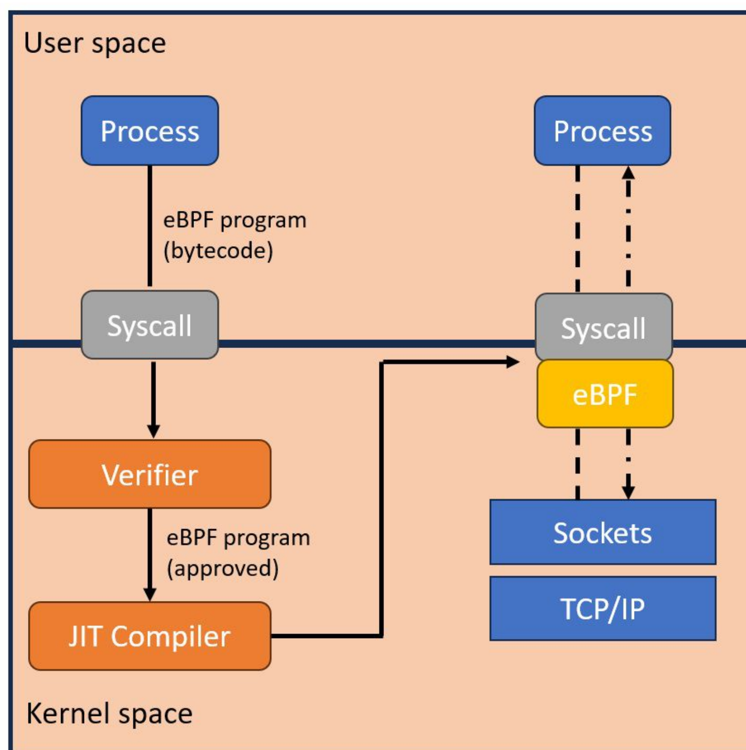

Figure 2.1: Scheme of the verification steps

2.1.3 Maps

One of the most important limitations of the cBPF is the lack of available memory. eBPF maps exceed this limit giving the developer a method to:

- Export data from kernel to userspace
- Data pushed by userspace to kernel
- Data shared between different eBPF programs
- Save eBPF program status (for stateful processing)
- keep state between kernel and user-space applications

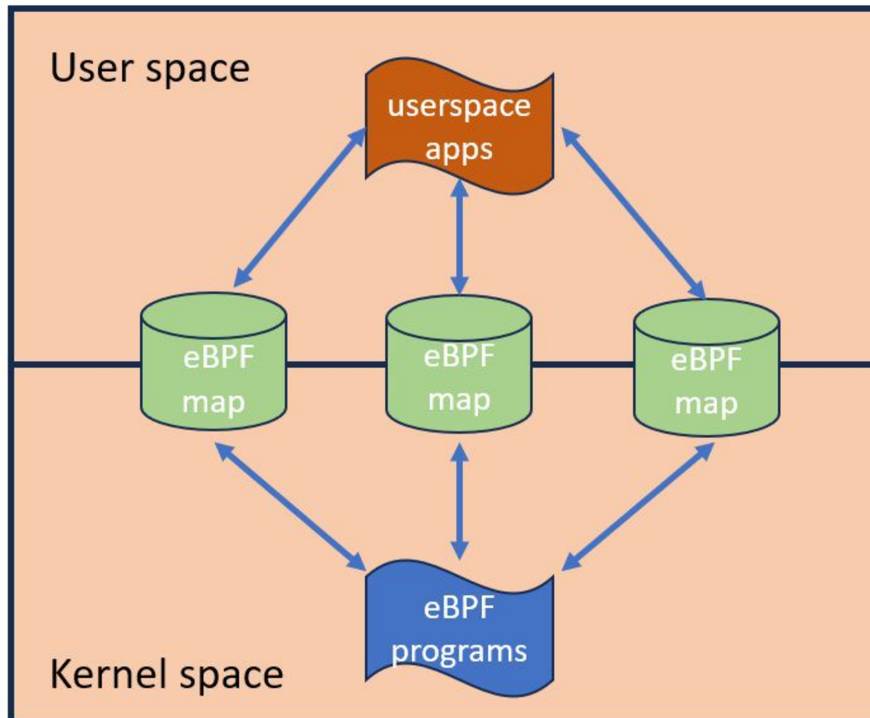

Figure 2.2: Interaction with eBPF maps

The different types of maps are defined in the enum `bpf_map_type` in include/uapi/linux/bpf.h. Some of them are described below:

- BPF_MAP_TYPE_ARRAY: this can be seen as an array. Key is an index and can only be 4 bytes. It is optimized for fastest possible lookup.
- BPF_MAP_TYPE_HASH: both user-space and eBPF programs can perform lookup, update and delete operations.
- BPF_MAP_TYPE_PERF_EVENT_ARRAY: this enables the transfer of unprocessed data to a performance ring, after which this data can be retrieved by a user-space program through buffer polling.

For the first two, there are also LRU and PERCPU versions. The LRU type

for handling the deletion of data from the table. The PERCPU mode allows you to link the map to a single running core to not bump into synchronization costs between the different cores by creating distinct memory areas.
eBPF maps can only be accessed through file descriptors. This was practical as the map file descriptor was readily available; however it is often limiting that a map can only be accessed from the same program that loads it. The mechanism for sharing BPF maps is called *pinning* [2]. The mechanism consists of creating a file for each map under a special file system mounted at `/sys/fs/bpf/`.

```
1 struct {
2     __uint(type, BPF_MAP_TYPE_ARRAY);
3     __uint(max_entries, 1);
4     __type(key, int);
5     __type(value, struct elem);
6     __uint(pinning, LIBBPF_PIN_BY_NAME);
7 } array SEC(".maps");
```

Listing 2.1: Definition of a pinned array map

2.1.4 Helpers

Another extension of the cBPF is the possibility to call a fixed set of in-kernel helper functions. These functions allow the program to interact with the system and with the context they are working with. Each program type (2.1.6) has its own subset of helpers. Some of the helpers are used to interact with maps (e.g. `void *bpf_map_lookup_elem(struct bpf_map *map, const void *key)`), while others can be used to manipulate network packets or for debugging [3].

2.1.5 Tail & Function Calls

eBPF programs can be chained together to create complex services. Communication happens through “tail calls”. From one eBPF program there is a jump into another eBPF program replacing the execution context without returning to the old one. This type of call is also implemented in eBPF to overcome the limited number of instructions per program. In order to execute tail calls, the `bpf_tail_call` helper must be invoked. The function takes as argument a BPF map that can hold references to BPF programs (BPF_MAP_TYPE_PROG_ARRAY). There is an upper limit to the number of successive tail calls for security reasons.

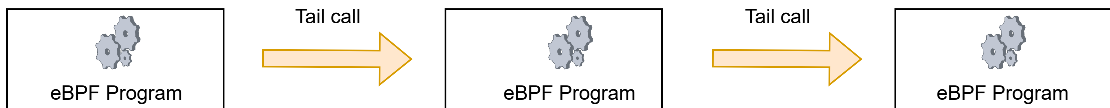

Figure 2.3: Tail calls between eBPF programs

2.1.6 Hook-points

eBPF programs are event-driven and are executed when the kernel or an application goes through a specific hook point. There are different types of eBPF hooks, including:

1. uprobes: userspace probes for tracing user space application functions

2. kprobes: dynamic kernel probes for tracing kernel functions

3. tracepoints: static probes for tracing specific kernel events

4. cgroup-bpf: hooks for applying eBPF programs to cgroups

For the part of packet processing there are two important program types: XDP (eXpress Data Path) 2.1.7 and TC (Traffic Control) 2.1.8.

2.1.7 eXpress Data Path (XDP)

eXpress Data Path (or XDP) provides a high-performance and programmable network data path in the Linux kernel [4]. XDP allows the developer to do a bare metal packet processing at the lowest point in the software stack. The possible use cases for this hook point are early packet discard (e.g. DDoS mitigation), forwarding, load balancing, and firewalling.
When the XDP hook point is triggered, the kernel has not performed resource-intensive operations yet, like allocating the socket buffer (SKB). The program takes as argument a struct xdp_md which has pointers to the beginning and to the end of the packet buffer, a pointer to a memory region to store additional metadata and indexes of the receive interface and receive queue.
The possible XDP actions are:

- XDP_PASS: the XDP program passes the packet to the normal network stack for processing. Even if the packet is sent to the normal stack, it may have been modified by the XDP program.
- XDP_DROP: it instructs the driver to drop the packet. It is the fastest way to drop a packet.
- XDP_TX: the packet is sent to the same interface it arrived on.
- XDP_REDIRECT: can be redirected to another interface or towards a map with the *bpf_redirect* or with *bpf_redirect_map* helpers.
- XDP_ABORTED: a program should never use this as a return code, it happens when an eBPF program has an error.

There are three ways to connect an XDP program to an interface:

1. Generic XDP: XDP programs are loaded as part of the normal network path. This is a way to test the program for drivers that do not provide support for XDP. Users can request this mode by setting the `XDP_FLAGS_SKB_MODE`.

2. Native XDP: it involves loading the program through the network card driver as part of its initial reception process. This mode necessitates specific driver support. Devices operating in Native XDP can only redirect packets to other devices also running in Native XDP mode. This approach offers greater efficiency compared to the previous mode.

3. Offloaded XDP: it entails loading the XDP program directly into the Network Interface Card (NIC), allowing it to execute without utilizing the CPU. This mode requires support from the network interface device and delivers even higher performance than Native XDP.

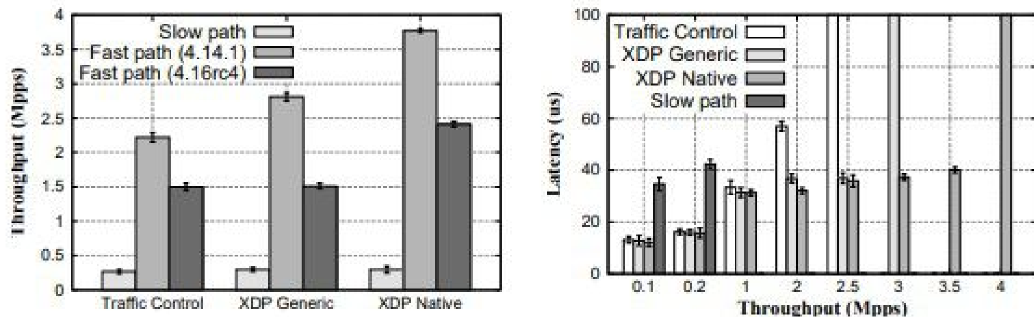

Figure 2.4: Comparison between XDP Generic and Native from [5]

2.1.8 Traffic Control (TC)

This program type is located in an upper point in the network stack (compared to the XDP). But this means having more pieces of information about the network

packet.
The input context is not anymore an `xdp_md` but it is a struct `__sk_buff`. After the XDP block, the packet passes through a layer that parses it and fills the buffer with metadata about the packet. From the TC ingress layer, the BPF program can use the metadata extracted from the packet. All members of the struct are defined in the `linux/bpf.h` system header. Some of the information is priority, protocol, vlan metadata, `tc_classid`.
As we can see in figure 2.5, TC program type can be in ingress and egress path while XDP is located in ingress only.

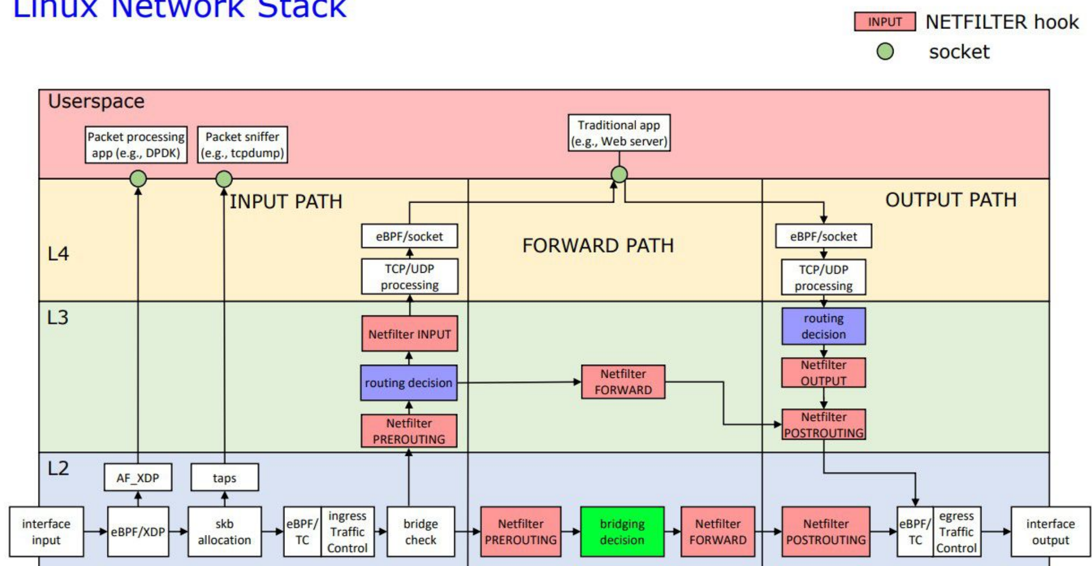

Figure 2.5: Linux Network Stack

2.1.9 Toolchain

eBPF programs are written in a restricted C code because as we said before (2.1.2), it is not possible to inject every kind of program in the kernel but it has to meet

some conditions checked by the action of the Verifier.
The CLANG frontend compiler and the LLVM backend are extended to accept the restricted C code. Furthermore, the LLVM core is used for general optimizations. So the compiler suite LLVM is used to compile pseudo-C code into eBPF bytecode. In previous kernel versions, the virtual CPU was implemented using an interpreter. However, in more recent versions, the bytecode can be compiled Just-In-Time (JIT) into native x64 code and pushed in the kernel.

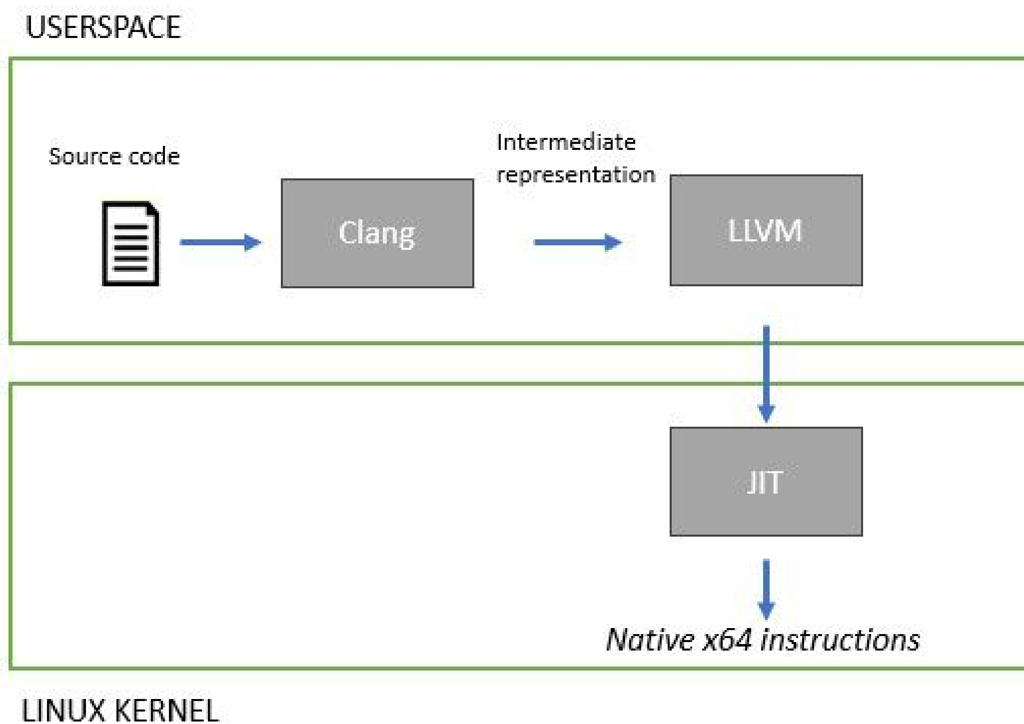

Figure 2.6: eBPF Toolchain

2.2 Traffic Control in the Linux Kernel

Tc is the tool to manipulate Quality of Service settings in the Linux Kernel [6]. It consists in:

- SHAPING: when traffic is subjected to shaping, its transmission rate is regulated. Shaping could involve more than just decreasing the available bandwidth. It is also used to handle bursts in traffic for better network behaviour. Shaping takes place on egress.

- SCHEDULING: with this method, it is possible to schedule the transmission of packets while still guaranteeing bandwidth to bulk transfers. The reordering of the packets is also called prioritizing and takes place only on egress.

- POLICING: this method occurs on ingress because it deals with the incoming traffic and not with the transmission of the packets.

- DROPPING: traffic that exceeds a set bandwidth can be dropped immediately, both on ingress and egress.

Qdisc

Qdisc, or queuing discipline, are one of the objects used in the traffic control. Every packet arriving on an interface is queued on the qdisc that has been set up for that interface. Afterwards, he kernel tries to get packets from the qdisc, for giving them to the network adaptor driver. There are two types of qdisc:

- Classless: The discipline of this type can't contain other qdiscs within their structure. They are solely responsible for tasks such as packet classification, delay, or dropping. Examples of classless qdiscs include FIFO (First In First Out), TBF (Token Bucket Filter), and SFQ (Stochastic Fairness Queueing).

- Classful: The discipline of this type have the ability to contain another qdiscs, which can be either classless or classful. This leads to the creation of a hierarchical tree structure of classes. Within this tree, internal nodes (including the root node) can hold other classful qdiscs, while the leaves contain classless qdiscs. Each classful qdisc discipline can also incorporate filters that determine the qdisc to which packets will be directed. As a result, packets traverse through this tree structure until they reach the leaves. An example of classful qdisc is the HTB (Hierarchical token bucket) qdisc, as well as PRIO. Some qdiscs, like HTB, allow for the dynamic addition of classes during runtime, while others, such as PRIO, are created with a fixed number of children. When a packet enters a classful qdisc, it can be categorized into one of the available classes based on various criteria, including tc filters, the Type of Service field, or the skb $\rightarrow$ priority set by userspace.

2.3 Packet Queuing and scheduling in XDP

Part of the thesis work focused on studying a recent patch [7] that has not yet been merged into the main Linux kernel. This patch aims to provide XDP with a mechanism for packet queuing. Indeed, XDP excels at forwarding packets quickly but it lacks the ability to reorder packets. This weakness makes impossible to use the XDP hook point for managing traffic while also applying traffic scheduling and traffic shaping policies. Since packet scheduling is an essential feature in modern networks, it is necessary to have these capabilities while continuing to leverage the speed of the XDP framework. This patch attempts to bridge this gap, providing an alternative to using the Queueing Discipline (Qdisc) that is part of Linux's traffic control system.

A crucial step in creating a customizable packet scheduling mechanism is having a flexible data structure that is compatible with the speeds of modern networks.

In recent years, the field of packet scheduling has seen significant development, as some generic data structures traditionally used for this purpose, such as binary heaps or red-black trees, no longer align with the requirements of modern networks.

2.3.1 PIFO and Eiffel extension

This patch has evolved around the Push-In-First-Out (PIFO) priority queue [8] and the Eiffel [9] extension to PIFO.
The PIFO structure originated in the hardware world, as it was designed to be integrated into switches for implementing scheduling algorithms. Subsequently, a software extension of this data structure, called Eiffel, was introduced. The PIFO data structure is a system composed of priority queues. Data can be queued at a specific priority level but can only be retrieved from the head of the structure. Although the structure may appear simple, it enables the implementation of a wide range of scheduling algorithms. The PIFO is a straightforward data structure characterized by the following attributes: it employs a ranking function based on integers to enqueue packets based on their priority, using a predefined range of ranks established during initialization; and it dequeues packets in accordance with the predetermined rank order.
The Eiffel extension is purely software-based and thus introduced several software-based optimizations. One optimization involves utilizing a bit-field to monitor queues with packets and leveraging the CPU's Find-First-Set instruction to expedite the search in the bit lookup table. Additionally, when the Eiffel system has numerous ranks, it can further divide the lookup table into a tree-based structure. Furthermore, it has introduced significant modifications to the original PIFO structure. Now, it can schedule not only individual packets but also entire flows, where each flow consists of a First-In-First-Out (FIFO) queue. Within the PIFO map, it can contain references to these FIFOs instead of just references to individual packets.

Moreover, in the original PIFO, the programmer could only perform scheduling on input, not on dequeue. This extension allows for the rearrangement of packets within the same flow by re-queuing them, thereby enabling the implementation of a broader range of scheduling algorithms.
The patch to the XDP hook introduces, in addition to the data structure, a mechanism for queuing and dequeuing packets. Two methods for implementing these parts are described in the following two sections.

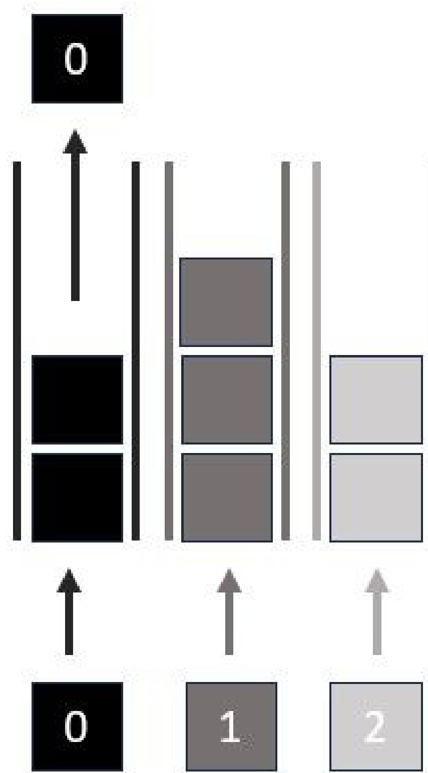

Figure 2.7: PIFO data structure: packets can be enqueued at a specific priority, but dequeue only occurs from the head.

2.3.2 First attempt: Adding a dequeue hook

The first attempt involves adding a new program type and a new hook point. The packet is redirected to the map using the existing helper function `static`

`long (*bpf_redirect_map)(void *map, __u32 key, __u64 flags)`, which is already available as an eBPF helper. Once the packet has been redirected, the transmission interface from which the packet should exit is activated using the helper function `static long (*bpf_schedule_iface_dequeue)(void *ctx, int ifindex, int flags)`. This is a new helper function. This function triggers a softirq during transmission, activating the program of type `xdp_dequeue`. This program retrieves the packet from the head of the PIFO and sends it to the outbound interface. In terms of code, an example of how it works is shown in listing 2.2.

```

struct {
    __uint(type, BPF_MAP_TYPE_PIFO_XDP);
    __uint(key_size, sizeof(__u32));
    __uint(value_size, sizeof(__u32));
    __uint(max_entries, 1024);
    __uint(map_extra, 8192); /* range */
} pifo_map SEC(".maps");


SEC("xdp")
int xdp_pifo(struct xdp_md *xdp)
{
    ret = bpf_redirect_map(&pifo_map, prio, 0);
    if (tgt_ifindex && ret == XDP_REDIRECT){
            bpf_schedule_iface_dequeue(xdp, tgt_ifindex, 0);
    }
    return ret;
}

SEC("xdp_dequeue")
void *dequeue_pifo(struct dequeue_ctx *ctx)
```

```
23 {
24     pkt = (void *)bpf_packet_dequeue(ctx, &pifo_map, 0, &prio);
25     if (!pkt){
26         return NULL;
27     }
28     return pkt;
29 }
```

Listing 2.2: Using the new dequeue hook and the new helper to schedule the transmission

This approach is the one used by Linux kernel maintainers so far to implement new features. However, this approach can be quite cumbersome, as introducing new program types and hook points requires writing a significant amount of code and makes code maintenance more challenging. For this reason, it was decided to attempt the approach presented in the following section, which is likely to be used for implementing future functionalities.

2.3.3 Second attempt: Using BPF Timer

This approach leverages BPF timers, introduced in Linux kernel version 5.15. In this approach, after redirection to the PIFO map, which always occurs within the XDP program, a timer is activated. Attached to this timer is a callback function that removes a batch of packets from the map. These packets are first sent to the outbound interface using the helper function `static long (*bpf_packet_send)(void *pkt, int ifindex, __u64 flags)` and then definitively ejected using the helper function `static long (*bpf_packet_flush)(void)`. In terms of code, an example of how it works is shown in listing 2.3.

```
1
2 struct {
3     __uint(type, BPF_MAP_TYPE_PIFO_XDP);
```

```
    __uint(key_size, sizeof(__u32));
    __uint(value_size, sizeof(__u32));
    __uint(max_entries, 1024);
    __uint(map_extra, 8192); /* range */
} pifo_map SEC(".maps");

static int xdp_timer_cb(void *map, int *key, struct bpf_timer *timer)
{
    for (i = 0; i < BATCH_SIZE; i++) {
        pkt = (void *)bpf_packet_dequeue_xdp(&pifo_map, 0, &prio);
        if (!pkt)
          return 0;

        bpf_packet_send(pkt, tgt_ifindex, 0);
    }

    bpf_packet_flush();
    return 0;
}


int xdp_pifo_timer(struct xdp_md *xdp)
{
    struct bpf_timer *timer;
    int array_key = 0;

    timer = bpf_map_lookup_elem(&array, &array_key);
    if (!timer)
        return XDP_ABORTED;

    if (!timer_init) {
        bpf_timer_init(timer, &array, CLOCK_MONOTONIC);
        bpf_timer_set_callback(timer, xdp_timer_cb);
```

```
37            timer_init = 1;
38      }
39
40      ret = bpf_redirect_map(&pifo_map, prio, 0);
41      if (tgt_ifindex && ret == XDP_REDIRECT)
42            bpf_timer_start(timer, 0 , 0);
43      return ret;
44 }
```

Listing 2.3: Using BPF timer to dequeue packets from PIFO map

Chapter 3

Prototype Architecture

3.1 General Architecture

The aim of the thesis is to implement network functions through the use of eBPF and the XDP block in such a way as to bypass the Linux kernel. The Linux kernel code is as generic as possible to handle every possible situation. On the other hand, eBPF is used for specific issues and conditions that need to be addressed. In this part of the work, attention is placed on the speeding up of the routing process and on the updating of its policies and on the implementation of a rate limiting mechanism.

3.1.1 Routing Acceleration

For the eBPF part of the **routing** process, a prototype developed in a previous thesis, also in collaboration with the Tiesse company, was used. A reactive approach rules this process. The first session packet, recognizable by the 5-tuple that identifies a specific TCP/IP connection (source IP address, destination IP address, source port, destination port and protocol), is not accelerated by the prototype but follows the normal network stack. In this way it is possible to save key parameters in specific memory areas, the maps. In this way, we will have as a result that only

the first packets of sessions follow a slower path, while the others, thanks to the information saved in the maps, are accelerated via software by the eBPF program. Given that pieces of information related to a specific session may change over time, it is essential to rely only on information saved in recent times. For this reason, an *age* field is also used to detect entries in the map that are too old and eliminate them. The *userspace* takes care of running a program that periodically cycles through all the entries and identifies those to discard. This solution has minimal impact on performance.
These features require placing the programs in two different hook points: one incoming on the XDP hookpoint, the other outgoing on the TC-egress block plus the program that runs in userspace. The XDP block is chosen because, as seen in section 2.1.7, it is the block that allows for the best performance by acting quickly on raw data that has not been processed yet. The other eBPF program has been placed in the TC-egress block, seen in section 2.1.8, since having reached that block we have all the necessary information available that can be used later for faster forwarding. The data is saved inside a hash map, pinned inside the *bpffs* virtual file system, to have a quick and simple use of its entries. Once the data is saved in the map, the packet can continue its journey through the network stack.
The XDP program uses the `bpf_map_lookup_elem(map_fd, void *key)` helper to figure out if data has already been saved for that session. If successful, it modifies the key parameters and redirects the packet to the correct output interface via the `XDP_REDIRECT` action; otherwise it allows the packet to go through the kernel via the `XDP_PASS` action.

Therefore the system uses a backlearning process to decide the output interfaces on which to redirect the packets. From figure 3.1 it is possible to see the execution flow of the part dedicated to speed up the forwarding process.

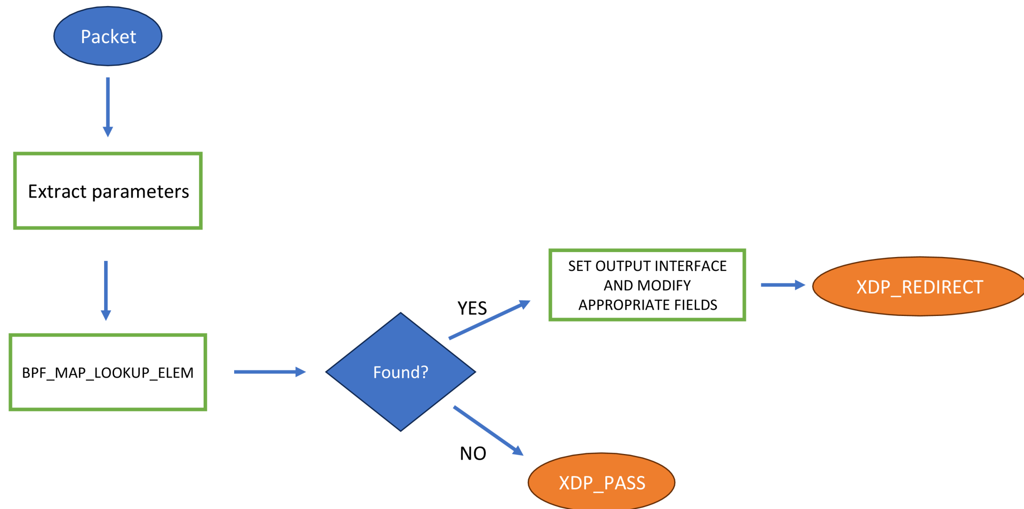

Figure 3.1: Flowchart for the routing process

3.1.2 Rate Limiter

Rate limiting is a technique used to control network traffic and prevent certain users or activities from consuming excessive system resources, making it useful for mitigating DoS attacks, among other purposes. There are two primary approaches to consider when implementing rate limiting [10]:

- TRAFFIC POLICING: it allows bursts of data to pass through. When the traffic rate reaches the configured maximum rate, any excess traffic is discarded. This results in an output rate that fluctuates over time. Traffic policing requires two key parameters: the average rate and the maximum burst size. The advantage is that it controls the output rate by dropping packets, avoiding delays caused by queuing. However, it can impact TCP window sizes and reduce the overall output rate of affected traffic streams.

- TRAFFIC SHAPING: In contrast to policing, traffic shaping retains excess

packets in a queue and schedules them for later transmission over specified time intervals. The outcome of traffic shaping is a smoother and more controlled packet output rate. It requires two parameters: the average rate and the burst size. Shaping relies on the presence of a queue and a buffer to temporarily delay packets. Additionally, it requires a scheduling mechanism for later transmission.

It is important to note that queues are primarily relevant for outbound traffic, where packets leaving an interface are queued and can be shaped. Inbound traffic on an interface, on the other hand, can only undergo policing.

Token Bucket Algorithm

Token bucket is an algorithm to configure a rate limiter. The tokens represent a unit of bytes or a single packet of a specific size. These tokens are added to a bucket at a fixed rate. So the algorithm requires two input parameters for each flow affected by rate limiting: the average bit rate and the maximum burst size. When a packet has to be sent, the bucket is checked if it contains sufficient tokens. If so, the right amount of tokens is removed from the bucket (the amount is equivalent to the length of the packet in bytes) and the packet is sent out. There are different actions that can be done with **non-conformant** packets:

- they may be dropped.
- they may be enqueued for subsequent transmission when the bucket will have enough tokens.
- they may be transmitted but marked as non-conformant.

The size of the bucket represents the maximum amount of bytes that tokens can be available at any given moment. The token bucket can be used in either traffic

shaping or traffic policing. In traffic shaping the packets are delayed, in traffic policing they are discarded or marked as non-conformant.

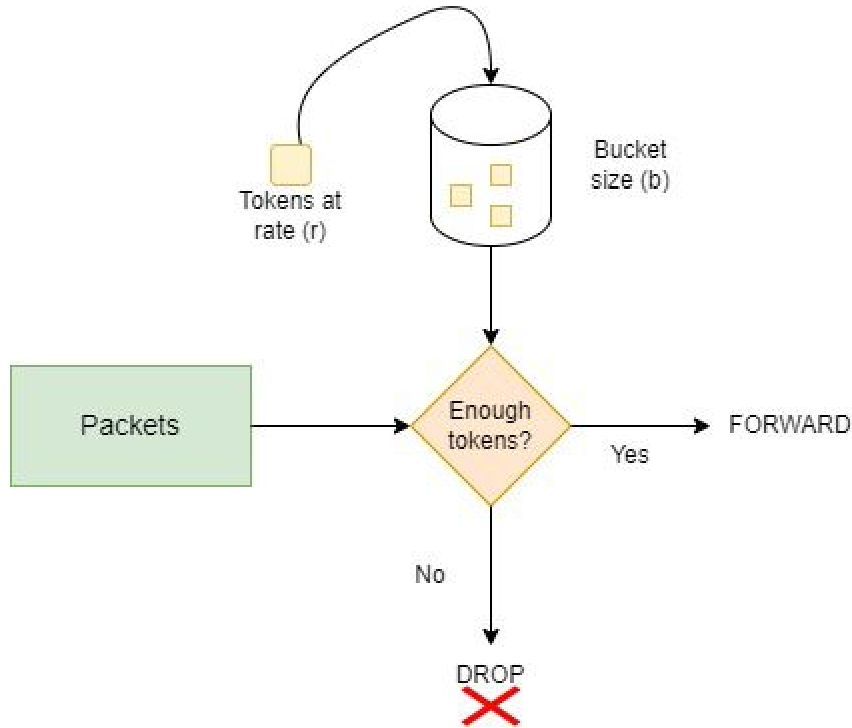

Figure 3.2: Token Bucket algorithm in traffic policing

The token bucket algorithm was used to insert the rate limiting part into the prototype. The realization was carried out with different techniques and methods made available by eBPF. In fact, the credit refill process can be found either in the XDP program or in a userspace program depending on the method followed. The realization and characteristics of each implementation will be illustrated later. Also for this rate limiting block, the **XDP** block is exploited because it is the one that guarantees the best throughput and the lowest CPU occupation in the face of a large workload.

3.1.3 Traffic policing and traffic shaping in the prototype

In section 2.2, the difference between traffic policing and traffic shaping is explained. The XDP framework currently does not include any packet queuing mechanism. The decision to forward or drop a network packet must be made immediately. For rate limiting functionality, this implies that only the traffic policing mechanism can be implemented in XDP. The prototype implementation presented in the subsequent chapter 4 follows the traffic policing strategy for implementing the token bucket algorithm.

The patch introduced in section 2.3 has the potential to address this gap. This means that it would be possible to incorporate a traffic shaping mechanism into the prototype for rate limiting management. However, the prototype's architecture at that point would require changes. Traffic subject to a traffic shaping rule must pass through the PIFO map before transmission. When dequeuing packets, the available tokens are checked to determine whether to forward the packet or not. Implementing this architecture may benefit from the use of circular buffers to store the sizes of queued packets in the buffer. This buffer would be consulted before dequeuing a packet from the map. If tokens are unavailable, it would be necessary to wait for the next callback invocation to dequeue the packets. The diagram for implementing traffic shaping is shown in figure 3.3.

Managing traffic shaping is significantly more complex than traffic policing. It involves handling a multitude of data structures that need to be instantiated for each rate limiting rule. One critical factor to test for such a scheme is the scalability that the system can achieve. It is also important to keep track of the presence of a high number of timers within the prototype. This could lead to excessive CPU overhead and a resulting degradation in performance. A critical point could be the management of the callback responsible for dequeuing packets. This program may need to go through long cycles for each PIFO map present within the prototype.

This slowdown could result in irregular flow throughputs.

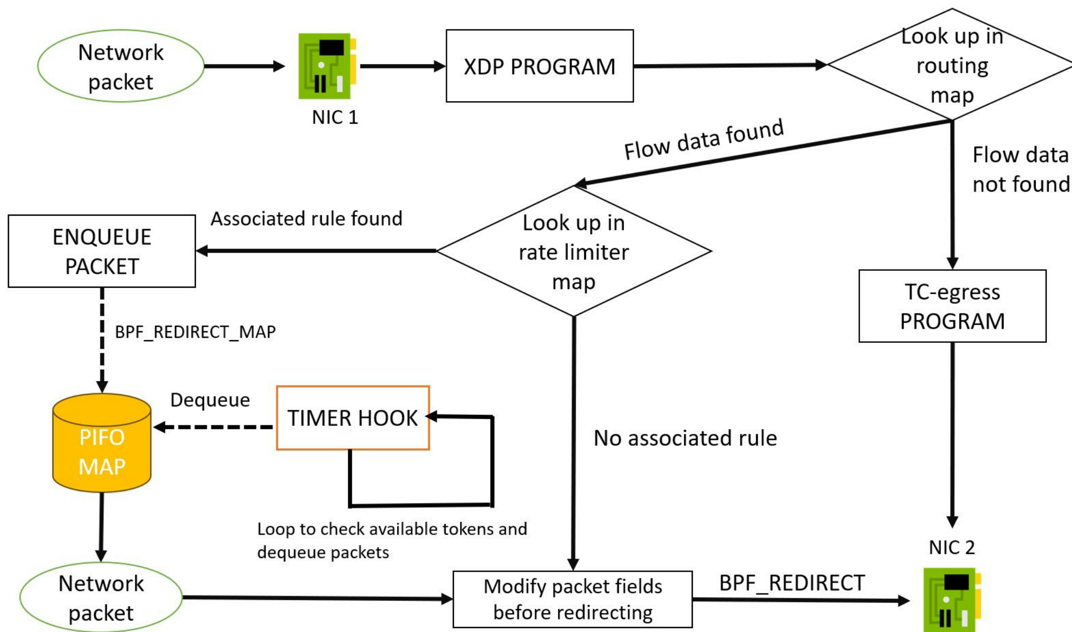

Figure 3.3: Flowchart for implementing traffic shaping

Chapter 4

Prototype Implementation

4.1 BPF skeleton

The life cycle of programs associated with the XDP hook and TC-egress hook is managed through the BPF skeleton. The BPF skeleton serves as an alternative interface to the existing libbpf APIs, specifically designed for working with BPF objects [11]. Its primary aim is to simplify the interaction with eBPF objects from the user space. The BPF skeleton achieves this by automatically generating customized code based on the input BPF object file, eliminating the need for manual component lookup. The resulting code accurately reflects the structure of the input BPF object, providing a comprehensive list of available maps, programs, and variables.

The interface facilitates access to **global variables** of various kinds (mutable, read-only, extern) and allows setting initial values before loading the BPF object. Generated code includes the embedded contents of the source BPF object file, ensuring synchronization between the skeleton and BPF object file.

Custom functions are generated with prefixes derived from the BPF object file name. These functions include opening and loading the BPF object, attaching and detaching BPF programs, and destroying the object to release resources. The

generated code and skeleton interface are designed to be interoperable with generic libbpf API.
Structs corresponding to global data sections, such as *.data*, *.bss*, *.rodata* and *.kconfig* are created for BPF objects with global variables. These structs facilitate setting initial values and updating data from userspace.
There is a set of functions in the skeleton in order to manage the lifecycle of the application [12]:

- <eBPFprog-name>__**create**_skeleton: to create the skeleton object starting from the object file coming from the compilation phase.
- <eBPFprog-name>__**destroy**: To dismantle the skeleton object, BPF maps are deactivated, and all resources utilized by the BPF application are released.
- <eBPFprog-name>__**open**: to open a file associated to the eBPF program. BPF object file is parsed: maps, programs and global variables are discovered, but not yet created.
- <eBPFprog-name>__**load**: to load a file linked to the eBPF program, BPF maps are established, and BPF programs are introduced into the kernel while undergoing verification. At this stage, all components of a BPF application are verified and exist within the kernel, yet no program is executed as of now. Following the loading phase, it becomes feasible to configure the initial state of BPF maps without encountering conflicts with the execution of BPF program code.
- <eBPFprog-name>__**attach**: to affix the loaded eBPF program, this phase involves attaching BPF programs to various hook points, such as kprobes or the network packet processing pipeline. During this phase, BPF programs become active and begin executing operations involving BPF maps and global variables.

- <eBPFprog-name>___**detach**: to detach an eBPF program and unload it from the kernel.

```
/* SPDX-License-Identifier: (LGPL-2.1 OR BSD-2-Clause) */

/* THIS FILE IS AUTOGENERATED BY BPFTOOL! */
#ifndef __PROTOTYPE_SKEL_H__
#define __PROTOTYPE_SKEL_H__

#include <errno.h>
#include <stdlib.h>
#include <bpf/libbpf.h>

struct redirect_semplice {
    struct bpf_object_skeleton *skeleton;
    struct bpf_object *obj;
    struct {
        struct bpf_map *xdp_map;
        struct bpf_map *index_map;
        struct bpf_map *counter_map;
        struct bpf_map *bss;
        struct bpf_map *data;
        struct bpf_map *rodata;
    } maps;
    struct {
        struct bpf_program *rate_limiter;
        struct bpf_program *xdp_redirect;
    } progs;
    struct {
        struct bpf_link *rate_limiter;
        struct bpf_link *xdp_redirect;
    } links;
    struct prototype__bss {
```

```
31        rate_limiter_rule entries[500];
32    } *bss;
33    struct prototype__data {
34        char __pad0[4];
35    } *data;
36    struct prototype__rodata {
37    } *rodata;
38
39 #ifdef __cplusplus
40    static inline struct prototype *open(const struct
     bpf_object_open_opts *opts = nullptr);
41    static inline struct prototype *open_and_load();
42    static inline int load(struct prototype *skel);
43    static inline int attach(struct prototype *skel);
44    static inline void detach(struct prototype *skel);
45    static inline void destroy(struct prototype *skel);
46    static inline const void *elf_bytes(size_t *sz);
47 #endif /* __cplusplus */
```

Listing 4.1: Example of eBPF skeleton header file

The generation of the skeleton goes through several phases. The eBPF source code is compiled using CLANG+LLVM. This generates the object file (.o). Then bpftool comes into play and through the command `bpftool gen skeleton file.obj` generates the skeleton file (.h). The phases to generate the skeleton are shown in figure 4.1.

4.2 TC-egress program

As mentioned in 3.1.1, the program located in TC-egress is triggered by the first packets of each session, when that `session_id` is not present in the map yet. The data comes in the form of the `struct __sk_buff`. The packet is examined at every

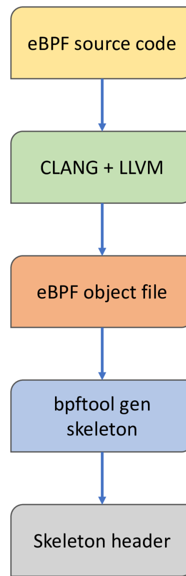

Figure 4.1: Flowchart of the eBPF skeleton generation

level of the TCP/IP stack, this is because the packet could be damaged and badly formatted due to physical layer transmission errors and therefore requires to be discarded. In order to do so, the buffer is first cast in the struct `ethhdr*`, then to the struct `iphdr*` and then, depending on the protocol, to the struct `tcphdr*` or `udphdr*`. Only the TCP and UDP protocols are handled by this accelerator, the others are passed through the kernel normally.
The map used for data exchange between TC-egress and XDP is a map of the `BPF_MAP_TYPE_HASH` type, called `xdp_map` (listing 4.2).

```
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __type(key, session_id);
    __type(value, _value);
    __uint(max_entries, MAP_SIZE);
    __uint(pinning, LIBBPF_PIN_BY_NAME);
} xdp_map SEC(".maps");
```

Listing 4.2: Definition of the xdp_map

Each entry of this memory area is formed by a key/value pair. In this project, the key used was a structure renamed `session_id` defined as follows:

```
typedef struct _key{
    __be32 saddr;
    __be32 daddr;
    __be16 sport;
    __be16 dport;
    __u8 proto;
} session_id;
```

Listing 4.3: Struct session_id

The value struct on the contrary is more complex and consists of:

- The first section consists of the package fields to be parsed to apply changes if required.
- The second part contains the fields necessary to forward the packet. It is possible to see the output interface and MAC addresses to be inserted before forwarding.
- The third section has the field AGE which represents the time passed from the creation of the entry in the map. It is necessary for cleaning old entries.

```
typedef struct fields{
    __u8 tos;
    __be16 h_vlan_TCI;

    __u32 ifindex;
    unsigned char    h_dest[ETH_ALEN];
    unsigned char    h_source[ETH_ALEN];

    __u64 age;
} _value;
```

Listing 4.4: Struct _value: contains all the fields necessary for accelerated network packet forwarding

After having "created" the key, it is therefore necessary to fill the fields of the *_value* structure. In order to copy MAC addresses, it is possible to use the __builtin_memcpy function. However, an assignment is sufficient for saving the output interface. To save the entry creation time eBPF provides the helper `BPF_KTIME_GET_NS()` which returns the number of nanoseconds passed since the system was started. The helper `bpf_map_update_elem(struct bpf_map *map, const void *key, const void *value, u64 flags)` is used to insert the entry in the map. The *flags* field can take on three values [3]:

- BPF_EXIST: The key's entry must be present in the map beforehand.
- BPF_NOEXIST: The key's entry must not be present in the map.
- BPF_ANY: There are no specific conditions regarding the existence of the key's entry.

4.3 XDP program

The XDP program is more complex as it includes both the part for optimized forwarding and the part for implementing rate limiting. The input structure to the XDP program is the struct `XDP_MD*` which contains very little information compared to the socket buffer.

```
/* user accessible metadata for XDP packet hook
 * new fields must be added to the end of this structure
 */
struct xdp_md {
    __u32 data;
    __u32 data_end;
    __u32 data_meta;
    /* Below access go through struct xdp_rxq_info */
    __u32 ingress_ifindex; /* rxq->dev->ifindex */
    __u32 rx_queue_index;  /* rxq->queue_index  */
    __u32 egress_ifindex;  /* txq->dev->ifindex */
};
```

Listing 4.5: xdp_md struct

Also in this case, as before, it is necessary to do all the checks to verify that the packet is formatted well and has not been fragmented incorrectly. In case there is some error the package is dropped via `XDP_DROP` action. After all the checks have been carried out, the key (the session id) to look at the map has also been obtained. In this way, it is possible to identify the flow in question. Using the `bpf_map_lookup_elem(struct bpf_map *map, const void *key)` helper, it checks if there is data for that session to proceed with a fast forwarding. The lookup in the map returns all the information needed to proceed. The time to live field (TTL) is decreased, the fields related to the type of service (TOS) and the belonging VLAN are modified and the source MAC address and the destination

MAC address are set.

```
_decr_ttl(ether_proto, l3hdr);
if(ret->tos != _val.tos)
    if(ip != NULL)
        ip->tos = ret->tos;
if(ret->h_vlan_TCI != _val.h_vlan_TCI)
    if(vhdr != NULL)
        vhdr->vlan_id = ret->h_vlan_TCI;

__builtin_memcpy(eth->h_source, ret->h_source, ETH_ALEN);
__builtin_memcpy(eth->h_dest, ret->h_dest, ETH_ALEN);
```

Listing 4.6: Packet mangling in XDP program

If the return value from the lookup function is *NULL*, it is necessary to proceed with the `XDP_PASS` action via the kernel. This decision is also made for a protocol other than UDP and TCP.
Now **rate limiting** comes into play. If the necessary information has been collected for that session, before proceeding with forwarding the packet, it is necessary to check whether there is a rate limiting rule associated with that flow. If not, you can forward the packet to the correct outbound interface with the `BPF_REDIRECT` helper. Instead, if so, it is necessary to consult a dedicated map to understand whether or not you have the tokens necessary for forwarding a packet of that size for that particular flow. Figure 4.2 shows the final scheme of the prototype which includes both the forwarding and rate limiting parts.

4.4 Rate Limiter different implementations

The implementation of the rate limiter has been tested with different strategies thanks to the tools that eBPF makes available to the developer. The rate limiter includes a part that is executed whenever a network packet arrives at a network

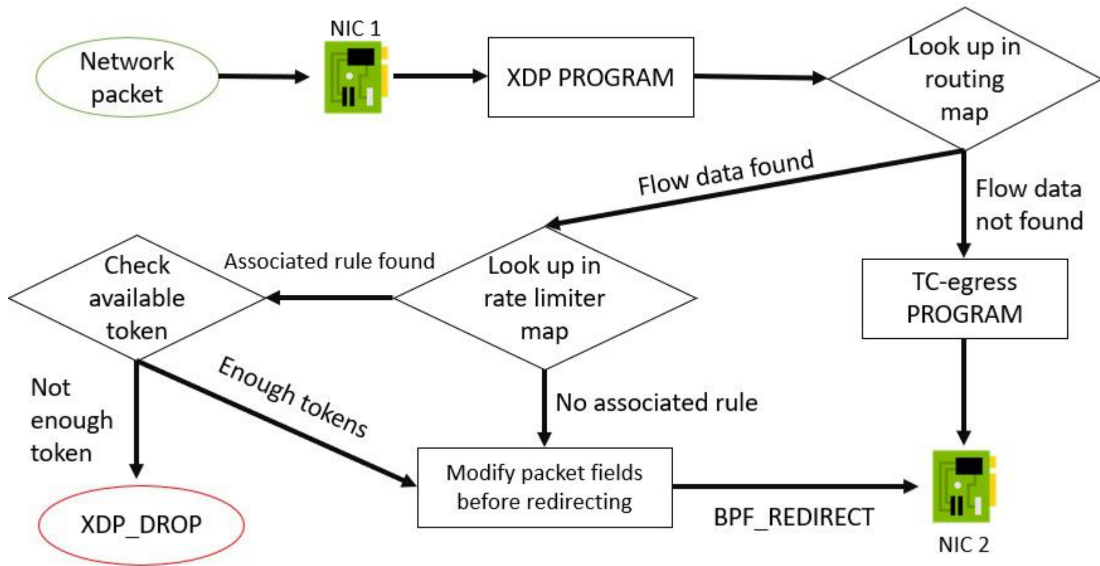

Figure 4.2: Flowchart of the entire prototype

interface. This part of the code, after having identified the flow, checks if for that flow the bucket has enough tokens to send the packet, considering the length of the packet itself.

The packet is dropped if the tokens are not enough in the bucket; otherwise, the packet is sent to the right output network interface and the available tokens of the bucket for that flow are decremented by the right amount.

This part is common between the various implementations of the rate limiter and can be seen in listing 4.7.

```
static inline int rate_limiter_apply(struct xdp_md *xdp, int *index){
    void *data = (void *)(long)xdp->data;
    void *data_end = (void *)(long)xdp->data_end;
    int packet_length = data_end-data;

    //Without this line, the verifier complains about
    //"math between map_value pointer and register with unbounded min
     value is not allowed"
```

```
8      //0x1F3 (499) is *index maximum value;
9      *index = *index & 0x1F3;
10
11     if((entries[*index].values_flows.tokens) >= packet_length ){
12         __sync_fetch_and_add(&entries[*index].values_flows.tokens,
13          -packet_length);
14         return bpf_redirect(tgt_ifindex, 0);
15     } else {
16         return XDP_DROP;
17     }
18 }
```

Listing 4.7: Rate limiter: token check and decision about the network packet

Line 10 in 4.7 is necessary as a workaround to not make the Verifier complain. In this case the Verifier wants to be sure that the index accessed in the array is between 0 and the length of the array.
This piece of code is included in the program inserted in the XDP block. After seeing the presence or absence of information for optimized forwarding as seen in section 4.3, a further lookup is performed in a different map which only collects the flows affected by rate limiting rules. If the lookup is not successful, the packet is redirected, while if it is successful, the function 4.7 is invoked for that packet.

```
1  int *ret2;
2  ret2 = bpf_map_lookup_elem(&index_map, &_key);
3
4  if(ret2 != NULL){
5      rc2 = S_SUCCESS;
6  } else {
7      rc2 = S_FAILED;
8
9  }
10
```

```c
switch(rc2) {
    case S_SUCCESS:
        return_value = rate_limiter_apply(xdp, ret2);
        if(return_value == XDP_DROP){
            return XDP_DROP;
        }
    case S_FAILED:
        return bpf_redirect(tgt_ifindex, 0);
    default:
        break;
}
```

Listing 4.8: Check if the flow is affected by a rate limiting rule

The loading of the rules happens through the reading of a file by a function in userspace. The parameters that must be passed to the program are the quintuple that identifies the session and the two fundamental data for rate limiting: the average rate and the capacity of the bucket.
For each rule, an entry is inserted in an array of `rate_limiter_rule structs`. This structure has all the data to enforce the rate limiting rules: the rate, the capacity, the tokens available for that flow, and the timestamp of the last refill made.

```c
struct values_flows {
    unsigned int refill_rate;
    uint64_t capacity;
    uint64_t tokens;
    unsigned long last_refill;
};


struct rate_limiter_rule {
    session_id session;
```

```
11     values_rules values_flows;
12 };
```

Listing 4.9: rate_limiter_rule and values_flows structs: used to determine the token state for each flow

The other fundamental part of implementing the rate limiting function is the token refill thread. To implement this function, different strategies can be adopted which are illustrated in the following sections. These solutions differ in the location of the function: in userspace or in kernel space. The diversity also consists in the way the function is periodically activated.

4.4.1 Refilling tokens using BPF Timer

The bpf timers have been introduced in the linux kernel since version 5.15. The BPF timer struct 4.10 can be embedded in *hash*, *array* and *lru* maps as regular field [13].

```
1  struct bpf_timer {
2      __u64 :64;
3      __u64 :64;
4  } __attribute__((aligned(8)));
5
6
7  struct {
8      __uint(type, BPF_MAP_TYPE_ARRAY);
9      __uint(max_entries, 1);
10     __type(key, int);
11     __type(value, struct bpf_timer);
12     __uint(pinning, LIBBPF_PIN_BY_NAME);
13 } array SEC(".maps");
```

Listing 4.10: BPF timer

In this prototype, an array map has been used to use the timer. BPF timer structure is based on the high-resolution timer API (*hrtimer*) already present in Linux Kernel. It is basically a wrapper of the hrtimer.
The setting of the timer goes through few stages and using some dedicated helpers:

- BPF_TIMER_INIT(STRUCT BPF_TIMER *TIMER, STRUCT BPF_MAP *MAP, INT FLAGS): this function initializes the timer. The first 4 bits of the 'flags' parameter specify the clockid, allowing only CLOCK_MONOTONIC, CLOCK_REALTIME, and CLOCK_BOOTTIME. All other bits in 'flags' are reserved. If the 'timer' is not associated with the same 'map', the verifier will reject the program. In user space, you should either have a file descriptor pointing to a map with timers or pin such a map in bpffs. When the map is unpinned or the file descriptor is closed, all timers in the map will be canceled and released.

- BPF_TIMER_SET_CALLBACK(STRUCT BPF_TIMER *TIMER, VOID *CALLBACK_FN): this function configures the timer to call the **callback_fn* function.

- BPF_TIMER_START(STRUCT BPF_TIMER *TIMER, U64 NSECS, U64 FLAGS): this function sets the timer's expiration time. The callback will be invoked in the soft irq context on some CPU, and if the timer is restarted, the next invocation may occur on a different CPU. This function increments the reference counter ('refcnt') of the BPF program to ensure that the callback code remains valid. When the user space reference to a map reaches zero, all timers in the map are canceled, and the corresponding program's 'refcnts' are decremented. This approach ensures that the termination of the user space process does not leave any timers running.

- BPF_TIMER_CANCEL(STRUCT BPF_TIMER *TIMER): this function cancels the timer and waits for the 'callback_fn' to finish if it was running.

In this version of the prototype, the token refill is performed in a callback linked to a timer which is initialized at the beginning of the XDP program. Some inaccuracies due to the actual duration of the timer have been corrected by calculating the elapsed time and calculating the correct proportion of tokens to add to the bucket.

```
static int xdp_timer_cb(void *map, int *key, struct bpf_timer *timer)
{
    unsigned long refill_rate, capacity, token_available;
    __u64 time_difference;
    int time_interval = 1000000;
    uint64_t refill_token;
    int nr_loops = nrule;  // Number of loops to execute
    #pragma clang loop unroll(full)
    for(int i = 0; i < nr_loops; i++) {
            refill_rate = entries[i].values_flows.refill_rate;

            if(refill_rate == 0){
                break;
            }
          unsigned long nsec = bpf_ktime_get_ns();

            time_difference = nsec - entries[i].values_flows.
    last_refill;

            if(time_difference >= time_interval){
                refill_token = refill_rate + ((refill_rate/100) * (((
    time_difference-time_interval)*100)/time_interval));
            }else {
                refill_token = (refill_rate/100) * ((time_difference
    *100)/time_interval);
            }

            capacity = entries[i].values_flows.capacity;
            token_available = entries[i].values_flows.tokens;

            if(token_available + refill_token >= capacity){
                refill_token = capacity - token_available;
            }

            __sync_fetch_and_add(&entries[i].values_flows.tokens,
    refill_token);
            entries[i].values_flows.last_refill = nsec;
        }
        return 0;
}
```

Listing 4.11: Refilling token callback

4.4.2 Refilling tokens using Perf Event

Another strategy that it is possible to follow to trigger an action periodically is to use *perf events*. The function 4.12 sets up a trigger based on the CPU clock by setting up an event based on `PERF_TYPE_SOFTWARE/PERF_COUNT_SW_CPU_CLOCK` [14].

Every time this value reaches a multiple of *freq*, the BPF probe will trigger and call the specified *prog*, which happens to be the refilling function, which is written in BPF code and is practically the same already shown in listing 4.11.

```
#include <bpf/bpf.h>
#include <bpf/libbpf.h>
#include <linux/perf_event.h>

static int open_and_attach_perf_event(int freq, struct bpf_program *
    prog, struct bpf_link *links[])
{
    struct perf_event_attr attr = {
        .type = PERF_TYPE_SOFTWARE,
        .freq = 1,
        .sample_period = freq,
        .config = PERF_COUNT_SW_CPU_CLOCK,
    };
    int i, fd;
    i = 0;

    fd = syscall(__NR_perf_event_open, &attr, -1, i, -1, 0);
    if (fd < 0) {
        if (errno == ENODEV)
            continue;
        fprintf(stderr, "failed to init perf sampling: %s\n",
            strerror(errno));
        return -1;
```

```
23    }
24
25    links_perf[i] = bpf_program__attach_perf_event(prog, fd);
26    if (!links_perf[i]) {
27        fprintf(stderr, "failed to attach perf event on cpu: %d\n", i
    );
28        close(fd);
29        return -1;
30    }
31    return 0;
32 }
```

Listing 4.12: Attach perf event from userspace

4.4.3 Refilling tokens from Userspace

In this strategy, the thread that deals with token refilling is in userspace. It is essential in this case to have a data access mechanism between userspace and kernel space. The helpers made available by eBPF do not guarantee this atomicity and risk making the refilling process inaccurate due to the continuous changes to the data coming from the XDP program.

A facility provided by BPF skeleton is an interface to mutable and read-only global variables. They are available starting from Linux 5.5 version. These can be used to pass data between userspace and kernel space and it is possible to use functions such as `__sync_fetch_and_add()` (which guarantees the atomicity of operations) also in userland. In cases where we frequently pass data back-and-forth between in-kernel BPF code and user-space, the usage of global variables can increase significantly performance: userspace can modify these variables directly without passing through bpf syscall [15]. Therefore the token refill thread is started

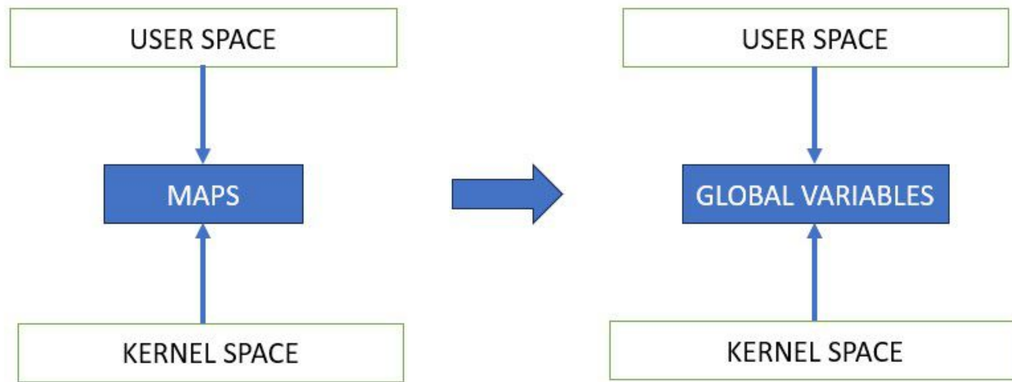

Figure 4.3: With token refilling performed by a userspace thread for more precise behavior, global variables were used instead of maps.

in userspace and periodically cycles on all the rules set, correctly updating the value of the tokens available for that flow. The logic of the refill strategy is very similar to the code seen previously in listing 4.11, so it is not reported for the sake of brevity.

Chapter 5

Prototype Evaluation

5.1 Testbed Setup

In this chapter, the results obtained from the acceleration prototype are presented. The tests it underwent include measurements of the maximum achievable system throughput and accuracy for the rate limiting component. The tests compare the results obtained from Linux kernel traffic control with XDP programs. Another crucial factor tested is the system's scalability as the number of flows increases. The test environment consists of two physical machines connected by two direct links, utilizing dual-port Intel XL710 40Gbps Network Interface Cards (NICs). The tester is used to generate network traffic that flows within the Device Under Test (DUT). The test prototype to be evaluated is loaded into the DUT.

- DUT: Intel Xeon Gold 5120 @ 2.20GHz processor with 28 cores and Ubuntu 20.04.6 LTS. The kernel version is 5.19.0.

- Traffic generator: Intel(R) Xeon(R) CPU E3-1245 v5 @ 3.50GHz processor with 8 cores and Ubuntu 18.04.6 LTS. The kernel version is 5.4.0.

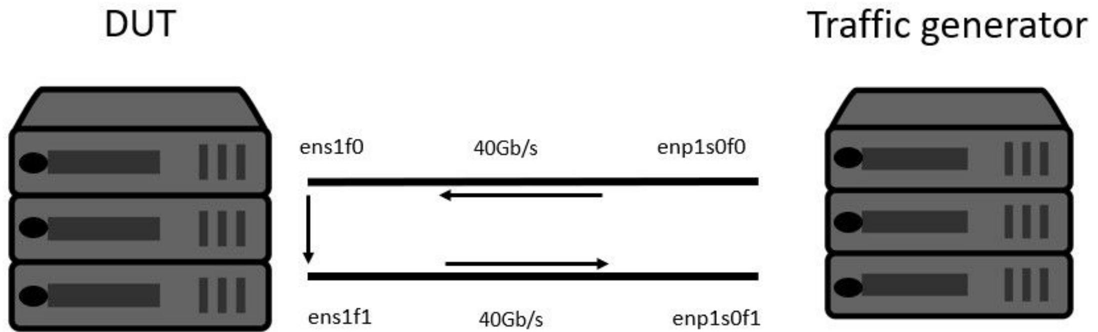

Figure 5.1: Testbed setup

5.2 Benchmarking tools

5.2.1 Iperf3

Iperf3 is an open-source network performance testing tool that is used to measure the maximum TCP and UDP bandwidth performance of a network. It allows users to assess the speed and efficiency of network connections by generating and measuring data transfer rates between two endpoints. For this thesis work, it was used to generate TCP traffic.

5.2.2 Cisco TRex

TRex [16] is an open-source, cost-effective traffic generator powered by DPDK, offering both stateful and stateless traffic generation capabilities. It can simulate L3-7 traffic and consolidate functionalities typically found in commercial tools.

TRex's stateless features encompass support for multiple streams, the ability to modify any packet field, and the provision of statistics, latency, and jitter on a per-stream or per-group basis. On the other hand, its advanced stateful capabilities enable the emulation of L7 traffic with comprehensive, scalable support for TCP/UDP.

To use TRex, the first step is to bind the relevant interfaces to DPDK. The configuration for this binding is passed through a YAML file where the IP address and default gateway for each interface are specified. Traffic management is then carried out through a Python file in which you can define the number of flows, packet size, and the percentage of bandwidth used. Additionally, it is possible to configure each field of the packet.

```
### Config file generated by dpdk_setup_ports.py ###

- version: 2
  port_limit: 2
  interfaces: ['0000:01:00.0', '0000:01:00.1']
  port_info:
      - ip: 1.1.1.1
        default_gw: 1.1.1.2
      - ip: 2.2.2.1
        default_gw: 2.2.2.2
```

Listing 5.1: Configuration of interfaces through a YAML file for TRex

For running TRex in server mode, you need to use the following command:

```
./t-rex-64 -i -c 6
```

Listing 5.2: TRex execution command in server mode

The -i option allows running the server in interactive mode, enabling connectivity via a console. The -c [N] option, on the other hand, allows you to specify the number of cores to be utilized for generating the desired traffic.

5.3 Routing tests

This test evaluates the routing part of the prototype. It is important to report these performance tests on the routing component because not all traffic is subject

to rate limiting rules. In addition, the numbers measured here represent an upper bound to the performance obtained when also QoS rules are set. Furthermore, even in case of QoS rules, some flows are free to utilize all the available bandwidth, so the maximum achieved throughput is also useful for this reason. In In this test, Linux forwarding is compared with the performance of XDP native and XDP Generic. In XDP Generic, XDP programs are loaded as part of the normal network path. This is a way to test the program for drivers that do not provide support for XDP. User can request this mode by setting the `XDP_FLAGS_SKB_MODE`.

The test involves a single UDP stream with a packet size ranging from 64B to 1300 bytes. Results are shown in figure 5.2. The bandwidth percentage was adjusted for each test to obtain only 1% packet loss. For all packet sizes, it is possible to see that XDP Native has the best performance, followed by XDP Generic and lastly Traffic Control. The XDP Native prototype achieves 5x better than Traffic Control performance and at least 3x better than XDP Generic. This happens because XDP generic simulates the behavior of native XDP, but in reality, the program execution occurs after the skb allocation, which is a very costly operation for the system. This test uses a single CPU core, that processes all the traffic and it is always 100% used.

5.4 Accuracy tests

The upcoming tests were conducted to evaluate the performance and accuracy of the prototype's rate limiting management component. Various approaches as described earlier (section 4.4) were compared. The behavior of XDP programs is juxtaposed with that of qdisc HTB, widely used within the Linux kernel. Within the tests, the traffic rate limit value will be varied to assess performance under lower or higher traffic conditions.

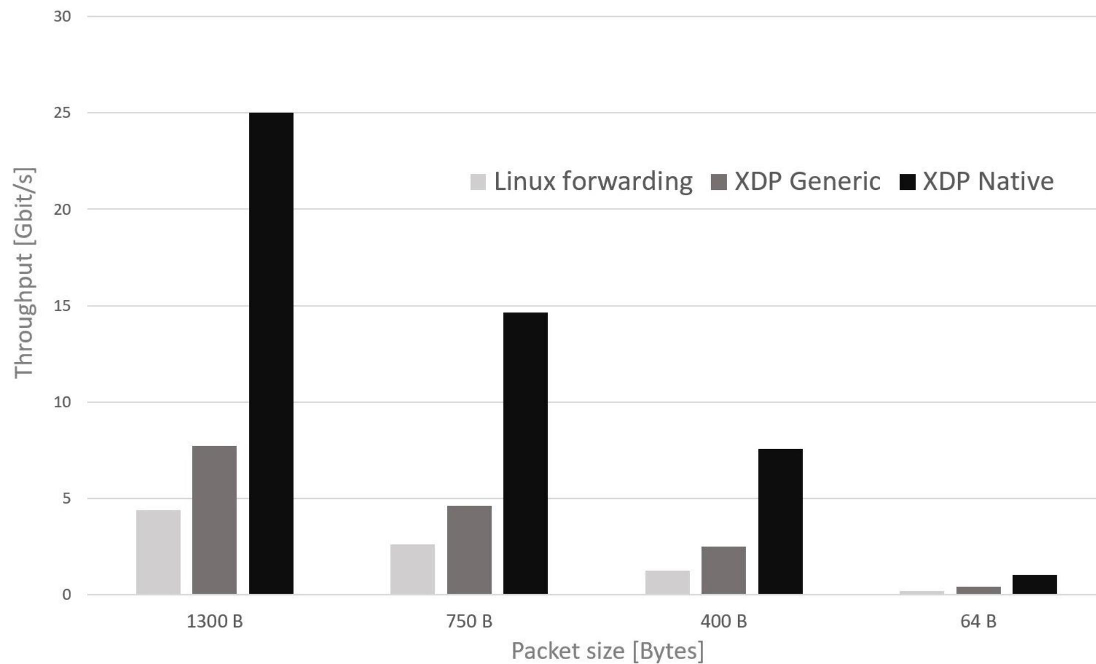

Figure 5.2: Throughput results varying packet size of one UDP flow

The HTB rule is set using the following script:

```
#!/bin/bash

UP_TOTAL_RATE="1000"
INTERFACE="ens2f1"

echo "Setting QoS parameters on interface $INTERFACE:"

sudo tc qdisc add dev $INTERFACE root handle 1: htb

echo -n "  Setting Queueing on interface $INTERFACE for an upstream
    traffic up to $UP_TOTAL_RATE kbps..."

sudo tc class add dev $INTERFACE parent 1:0 classid 1:12 htb rate ${
    UP_TOTAL_RATE}Mbit ceil ${UP_TOTAL_RATE}Mbit

```

```
14 echo "done."
15
16 echo -n "Setting filters... "
17
18 sudo tc filter add dev $INTERFACE parent 1:0 protocol all prio 0 u32
    match ip src 1.1.1.1 match ip dst 2.2.2.1 match ip sport 1000 0
    xffff match ip dport 2000 0xffff match ip protocol 17 0xff flowid
    1:12
```

Listing 5.3: Configuration of HTB rule for an interface

For XDP programs, on the other hand, rules are set by specifying the session that should be affected by the rule and the traffic rate that must be adhered to. When the programs are loaded, this file is read, and the rules are validated and loaded into their respective maps, also initializing the available token values.

The tests were conducted for both TCP and UDP. The values at which the flows are ‘cut off’ range from 200 Mbit/s to 1 Gbit/s. The results are presented in 5.3 and 5.4. The results exhibit very similar behavior in terms of accuracy, indicating that all precautions taken for synchronizing token changes are effective. There is a noticeable difference in core utilization where traffic is received. Figure 5.5 shows the core utilization percentage for UDP traffic at 600 Mbit/s while varying packet sizes. It can be observed that HTB fully occupies the core in all cases, whereas for XDP programs, the perf event strategy consumes fewer resources.

5.4.1 Comparison among the various approaches presented

To implement the refilling function, different strategies have been adopted as illustrated in the previous sections. As we said, these solutions differ in the location of the function: in userspace or in kernel space.

The level at which programs are executed varies the way we can write them:

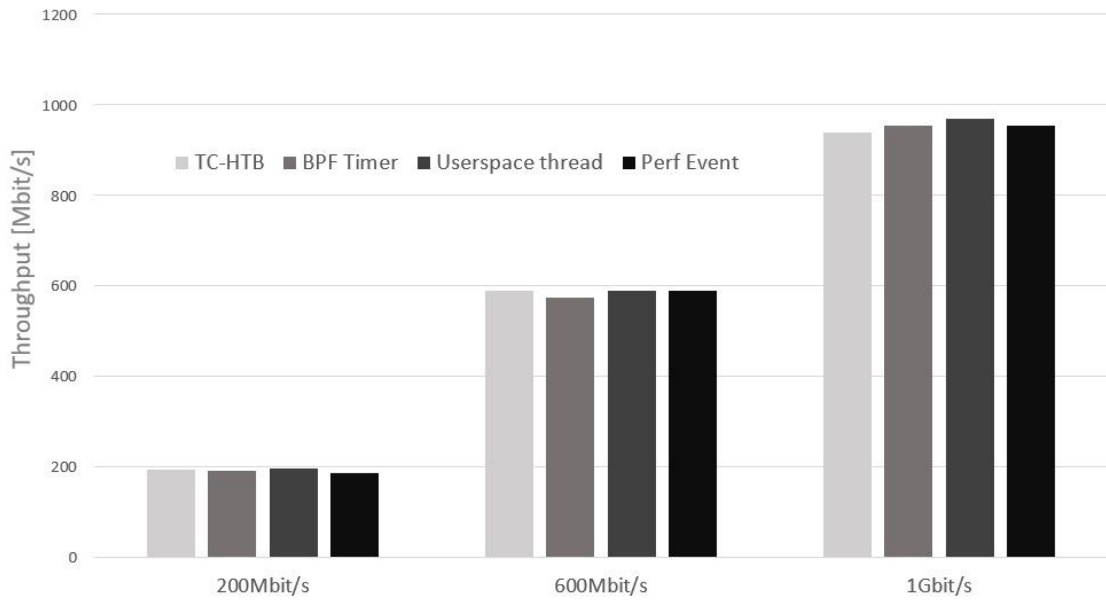

Figure 5.3: Accuracy test with TCP while varying the rate to be adhered to for the flow

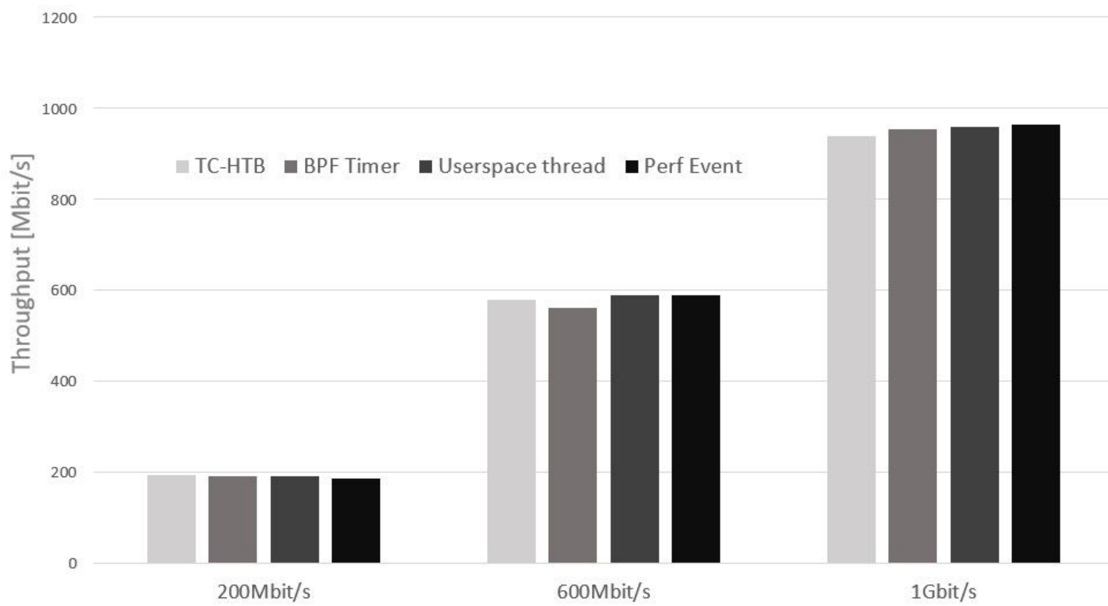

Figure 5.4: Accuracy test with UDP while varying the rate to be adhered to for the flow

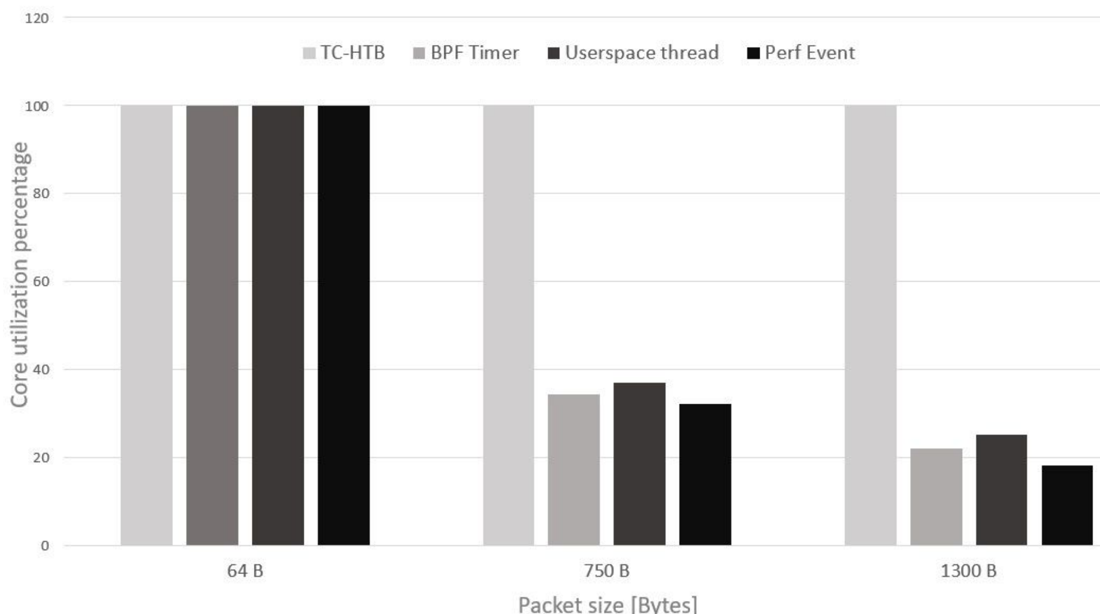

Figure 5.5: Occupancy of the core to which traffic is redirected varying packet size - UDP flow

programs written in eBPF and run in kernel space must comply with the rules imposed by the verifier. Whereas those running in userspace have more freedom and pose fewer issues during compilation. Especially if you are constrained by an older version of the Linux kernel, the checks imposed by the verifier can lead to many problems, such as the acceptance of loops within the code.

Furthermore, another factor to consider is the resource utilization due to the various strategies. As can be seen from Figure 5.5, different implementations exhibit differences in the core occupancy receiving the traffic. The perf event strategy seems to be the best in terms of resource utilization for triggering an event at a specific frequency. BPF timer could be an acceptable solution, but it is important to consider that having a multitude of timers in kernel space can lead to system slowdowns. The userspace thread appears to have higher resource utilization in this case because both kernel space and userspace are trying to access the same data. The need to synchronize the two accesses and the frequent context

switches contribute to this increased overhead.

5.5 Scalability test

Scalability is a critical factor to test in the prototype. In this case, we chose to test it by increasing the number of UDP flows sent to the system, with each flow being subject to a rate-limiting rule. This rule, in reality, does not actually cut off the passing traffic; it is set at the maximum link speed (in this case, 40 Gbit/s). This way, the measured packet rate is at its maximum, and the system still needs to manage credit refilling for all the flows entering the DUT.
The results are presented in Figure 5.6. It can be observed that with this number of flows, the system does not appear to have weaknesses or be adversely affected. In this test, all available cores are utilized thanks to Receive Side Scaling (RSS), which distributes the flows across different cores.

5.6 LS1046A Freeway Board

The LS1046A Freeway board [17], also known as LS1046A-FRWY and produced by NXP, a company based in the Netherlands, is a high-performance computing, evaluation, and development platform. It is powered by the LS1046A architecture processor, capable of delivering over 32,000 CoreMark performance. The board features onboard DDR4 memory, multiple Gigabit Ethernet ports, USB3.0 support, and M.2 Type E interfaces for Wi-Fi connectivity. Additionally, the FRWY-LS1046A-AC variant includes a Wi-Fi card. This platform is equipped with an accelerator for Machine Learning and Artificial Intelligence applications.

The prototype was also loaded and tested on this board, which was provided by Tiesse. The construction of the file system and all the components for its operation

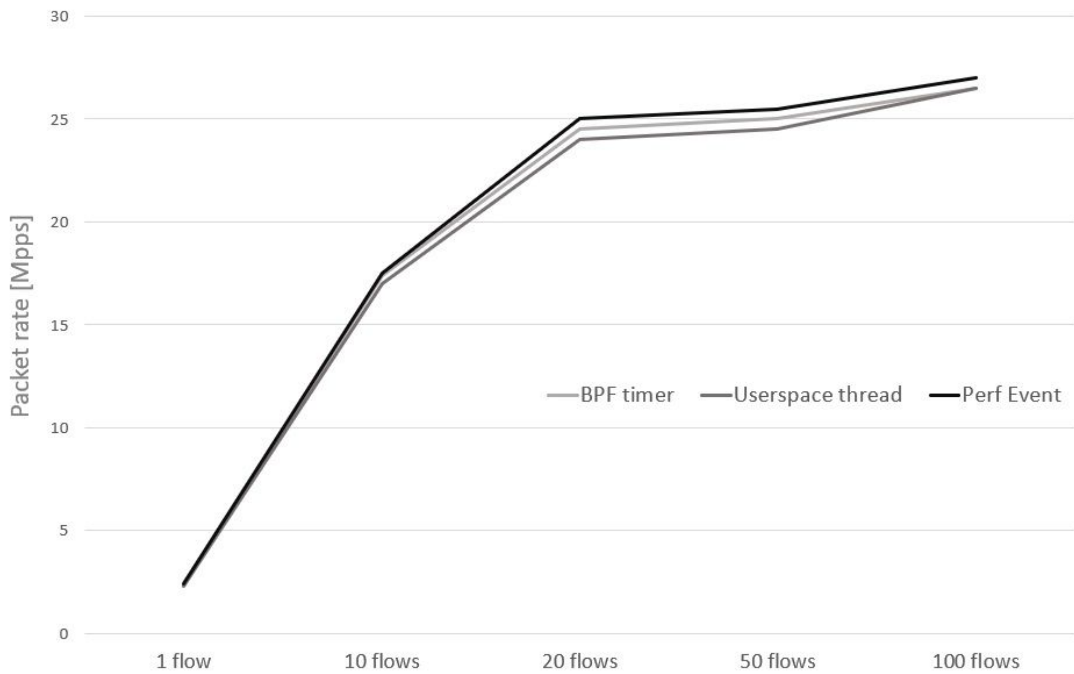

Figure 5.6: Scalability test - measuring packet rate increasing number of UDP flows

were managed using the flex-builder program.
At the beginning, the patched version of the Linux kernel provided by NXP that was installed on the device was version 5.4.3 aarch64. However, this kernel version made it impossible to load this type of prototype. BPF timers became usable starting from Linux kernel version 5.15, and it was not possible to load perf event programs on this board. Therefore, the option of userspace thread for token refill remained, but there were also issues encountered here: the use of global variables through the skeleton was only possible with a version higher than 5.10.0 of the kernel. Thus, the entire device configuration process was repeated by installing Linux kernel version 5.10.35 aarch64. At that point, it became possible to load the prototype using the userspace thread refill strategy. Furthermore, the network card drivers on the board do not support XDP Native, so the programs are loaded as XDP Generic.

To test the prototype on the board, the testbed setup shown in figure 5.7 was used.

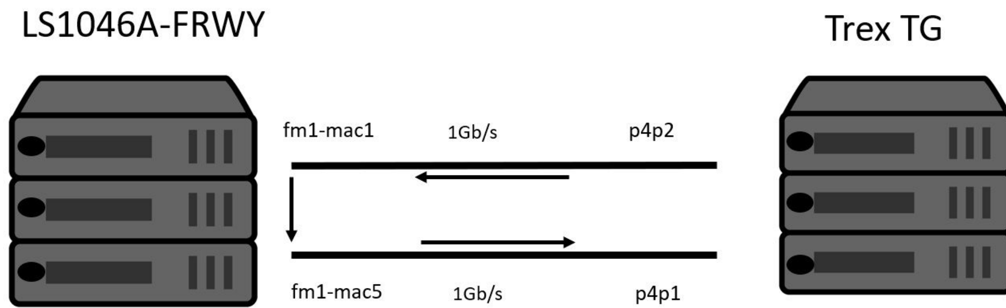

Figure 5.7: Testbed setup used to test LS1046A Freeway Board

In this case as well, the Linux kernel's traffic control system was compared to the XDP prototype. Figure 5.8 depicts the routing test with a UDP flow generated by Trex, varying the packet size. The graph shows the throughput obtained in both cases. With large packets, the difference is minimal. However, this difference starts to increase as the packet size decreases, reaching almost three times higher throughput with 64B packets. During the test, the condition of the board's cores was also monitored. The results are shown in Figure 5.9. The advantage is evident with all packet sizes. With 64B packets, resources are fully utilized, while the XDP prototype leaves 26% of the resources idle.

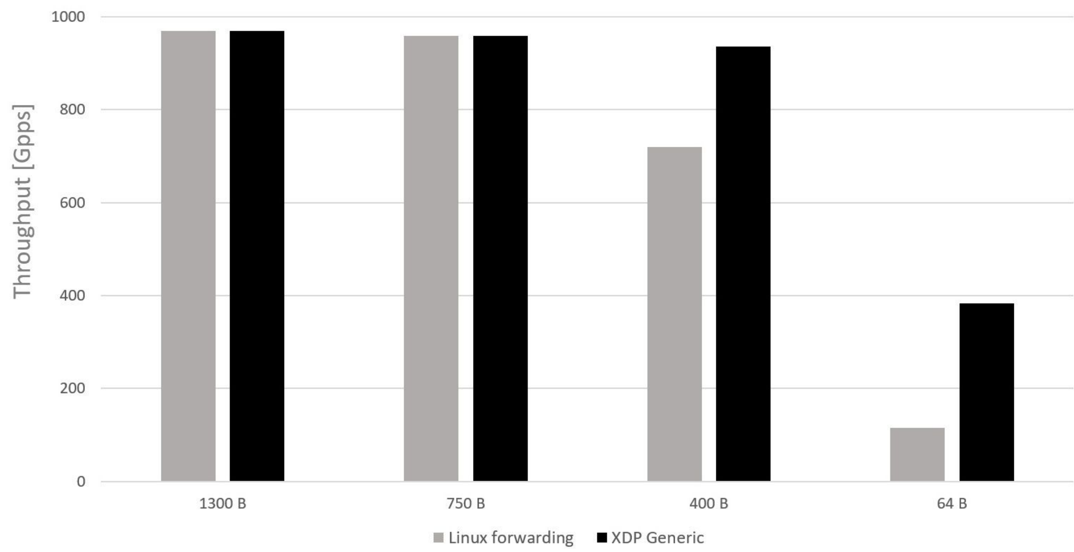

Figure 5.8: Throughput result varying packet size of one UDP flow - LS1046A Freeway Board

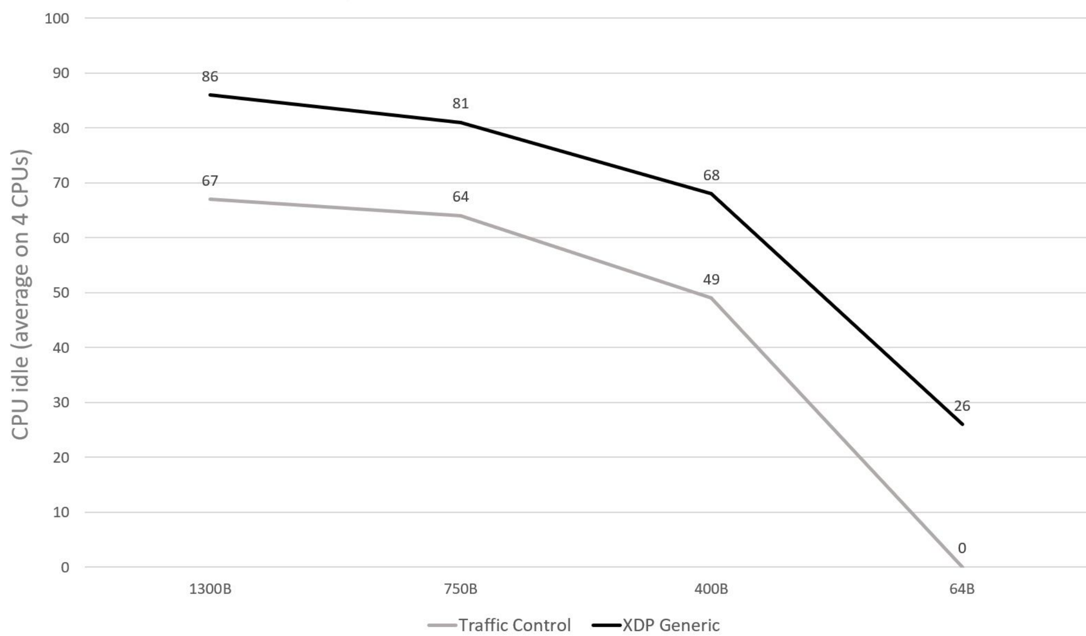

Figure 5.9: Percentage of CPU idle varying packet size of one UDP flow - LS1046A Freeway Board

Chapter 6

Benchmarking XDP scheduling with PIFO map

In this chapter, we will evaluate the performance and capabilities of the patch presented in Section 2.3. The patch compares the behavior of XDP with these new features to vanilla XDP. Additionally, various queuing methods are introduced and compared with the Qdiscs in the traffic control system that perform similar functions. We will consider the total throughput of flows and the core utilization on which the traffic is received.

6.1 Baseline impact of the PIFO map

The testbed setup is the same as presented in Section 5.1 and is not reiterated here for brevity. The objective of this test is to understand how passing through the PIFO map impacts the total throughput. In this case, we compare the Linux traffic control system's bridge, XDP redirection without passing through the map (vanilla XDP), and XDP redirection with passing through the map. The method for scheduling transmission involves using the callback attached to the `bpf_timer`. The priority at which packets are queued here is set to 0 for all, so the map is used as a simple FIFO.

The test is conducted using a UDP flow while varying the packet size. As evident from the results in figure 6.1, vanilla XDP redirection maintains a throughput 1.5 times higher than XDP scheduling. Nevertheless, XDP scheduling exhibits a throughput up to 8 times greater than the traffic control bridge. The utilization of the core receiving the traffic is at 100% in all cases.

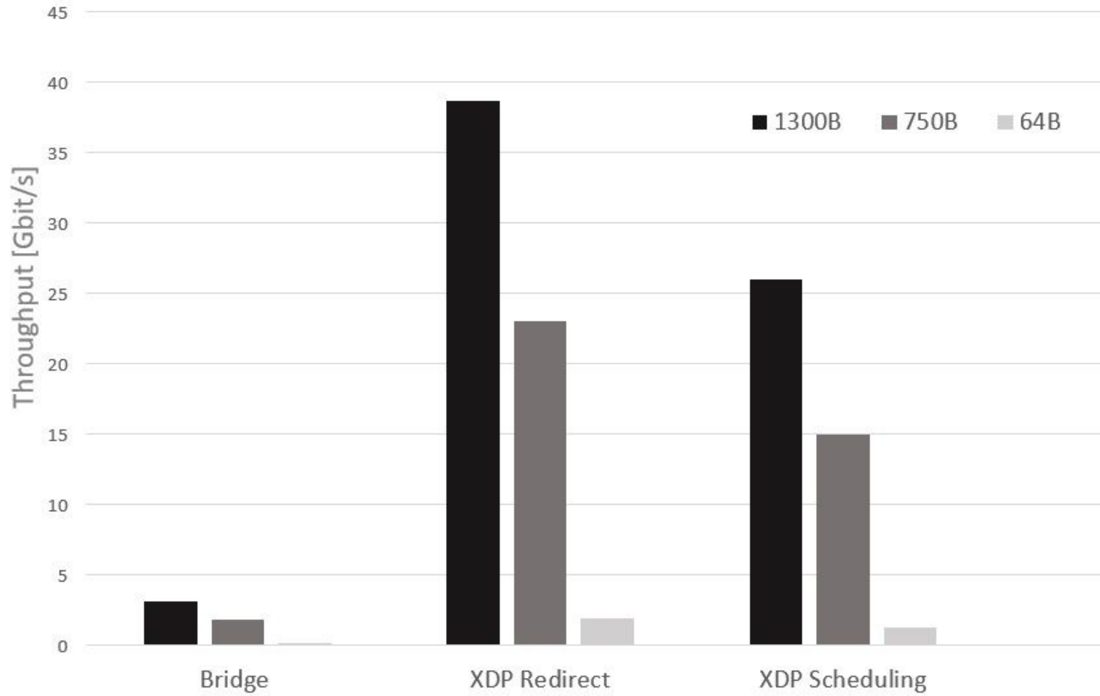

Figure 6.1: Comparison between bridge, vanilla XDP Redirect and XDP Scheduling(packet passes through PIFO map) - One UDP flow varying packet size

The test was also conducted with a single TCP flow. Figure 6.2 presents a dual-axis graph depicting both the total throughput and the core utilization percentage. In this test using iperf3, with vanilla XDP redirection, it is observed that the traffic reaches 27 Gbit/s, and the core is not fully utilized. This suggests that the throughput is limited by iperf3 and not by the capabilities of XDP redirection, which could achieve better results. Once again, XDP scheduling exhibits a slowdown compared to vanilla redirection but maintains a significant advantage over the

traffic control bridge.

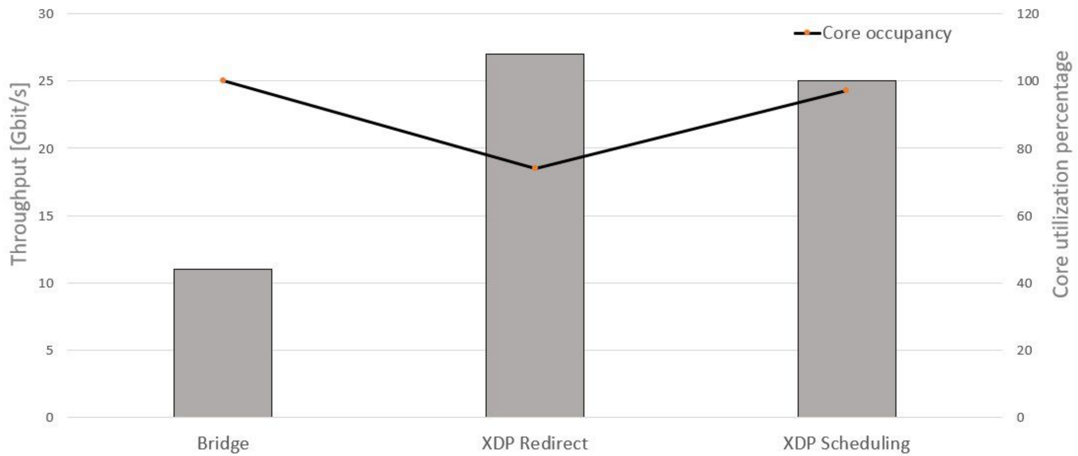

Figure 6.2: Comparison in throughput and core utilization between bridge, vanilla XDP Redirect and XDP Scheduling(packet passes through PIFO map) - One TCP flow

6.2 Strict priority

The strict priority scheme dictates that traffic with higher priority is sent before all others. Lower-priority queues are only scheduled after the high-priority queues have been serviced. This comparison is performed between the XDP program and the PRIO Qdisc.

The PRIO qdisc [18] is a straightforward queuing system organized into various priority classes. These classes are dequeued in descending numerical order based on their priority. PRIO operates as a scheduler and does not introduce delays to packets; it is a work-conserving qdisc, even though the qdiscs within the classes it contains may not follow the same principle.

When you create it using 'tc qdisc add,' you specify a fixed number of bands (classes) to be created. While you cannot add classes using 'tc qdisc add', you must

indicate the number of bands you want to create when attaching PRIO to its root. During dequeuing, PRIO first attempts to dequeue from band 0. Only if band 0 does not have any packets will PRIO proceed to band 1 and so on. Typically, high-reliability packets should be directed to band 0, those requiring minimal delay to band 1, and the remaining to band 2.
PRIO offers three different approaches to decide which band a packet will be placed into:

- Using a tc filter: You can attach a tc filter to the root qdisc, which allows you to directly route traffic to a specific class.
- From userspace: If a process has the necessary privileges, it can directly specify the destination class
- Using the priomap: This method relies on the packet's priority, which is derived from the Type of Service assigned to the packet. The priomap is based on this packet priority to determine its class placement.

```
#!/bin/bash

INTERFACE="ens2f1"

sudo tc qdisc add dev $INTERFACE root handle 1: prio bands 2 priomap
    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

sudo tc qdisc add dev $INTERFACE parent 1:1 handle 10: pfifo
sudo tc qdisc add dev $INTERFACE parent 1:2 handle 20: pfifo

sudo tc filter add dev $INTERFACE parent 1:0 protocol all prio 0 u32
    match ip dport 5001 0xffff flowid 1:1

```

```
12 sudo tc filter add dev $INTERFACE parent 1:0 protocol all prio 1 u32
     match ip dport 5002 0xffff flowid 1:2
```

Listing 6.1: Configuration of PRIO Qdisc: traffic to port 5001 has higher priority compared to traffic to port 5002

In the XDP program, rules are inserted into a text file by specifying the session identifier and the priority value assigned to the flow (a smaller number corresponds to a higher priority). This value is then stored in a HASH map, with the session identifier as the key and the priority at which to enqueue the packet as the value. The code snippet below 6.2 demonstrates the lookup within the map, from which we obtain an integer indicating the priority at which to enqueue the packet in the PIFO map.

```
1  int *ret2;
2  ret2 = bpf_map_lookup_elem(&index_map, &_key);
3
4  if(ret2 != NULL){
5      prio = *ret2;
6      break;
7  } else {
8      prio = 0;
9  }
10
11 ret = bpf_redirect_map(&pifo_map, prio, 0);
```

Listing 6.2: Determination of the priority at which to enqueue the packet within the PIFO map

The test to compare scheduling in XDP and the PRIO qdisc was carried out by generating two TCP flows and, as seen above, assigning maximum priority to one of them. Traffic from both flows is handled by a single core.

The setup used here 6.3 is slightly different; a bottleneck was introduced because,

with interfaces at the same speed, the rules of the qdisc would have no effect. Rules come into play when there is a lack of resources. Traffic starts from a 40Gbit/s interface and terminates at a 10Gbit/s interface.
From Figure 6.4, it can be seen that the resulting throughputs of the two flows are very similar: the higher-priority flow occupies almost all available bandwidth, leaving a small percentage of bandwidth for the lower-priority flow. However, it is essential to consider the core utilization that manages the two flows: in the case of the PRIO qdisc, it is fully occupied, while in the other case, it is 45% occupied. This demonstrates how a strict priority mechanism operated by the XDP block can achieve the same results as the existing qdisc and leave system resources much more available.

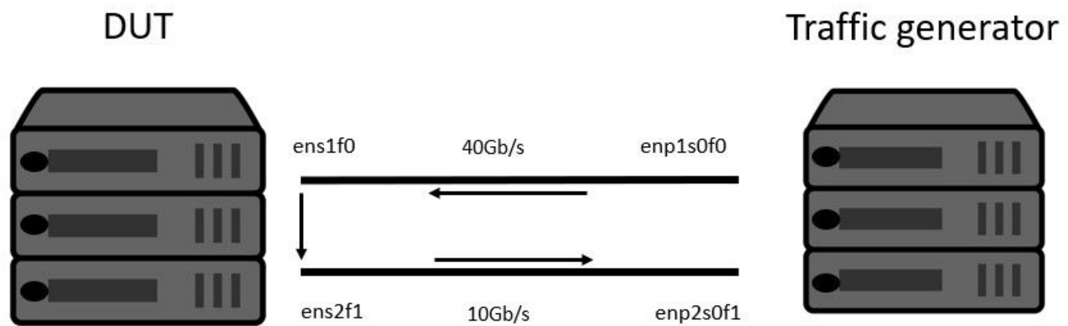

Figure 6.3: Testbed setup used in the test for strict priority: DUT acts as a bottleneck

6.3 Weighted Fair Queueing

Weighted Fair Queueing (WFQ) is a scheduling algorithm used in computer networks to manage the flow of data packets. This algorithm assigns a weight to each packet queue and then distributes outgoing packets based on these weights. This way, queues with higher weights receive a larger share of network resources compared to others. It is used in networks with differentiated services, to ensure that services

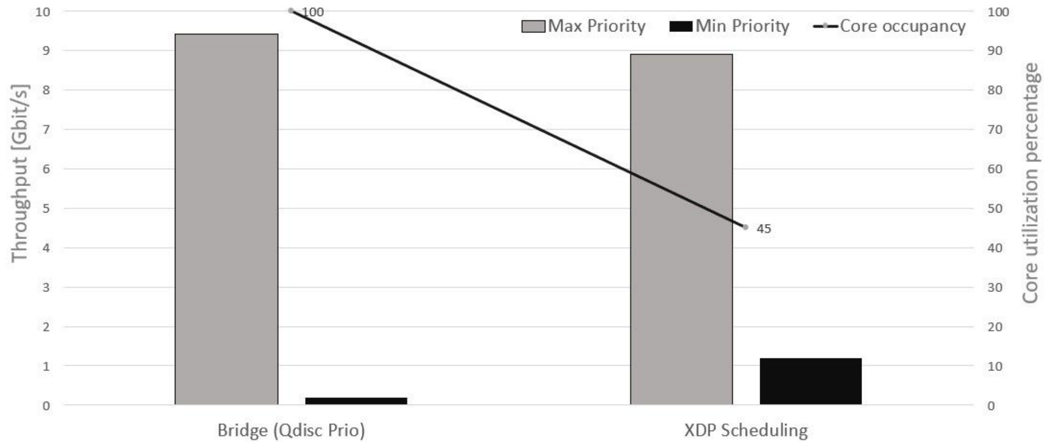

Figure 6.4: Comparison in throughput and core utilization between PRIO Qdisc and XDP program - 2 TCP Flows redirected to one core

with different latency or bandwidth requirements are managed fairly or weighted appropriately.

The comparison is now made with the HTB qdisc, which allows for this type of scheduling algorithm. HTB [19] is designed to help you manage the outbound bandwidth on a specific network link. It allows you to emulate multiple slower links using a single physical link and route different types of traffic through these emulated links. In both cases, you need to specify how to divide the physical link into simulated links and determine which simulated link to use for a given packet. Within a single HTB instance, there can be multiple classes. Each of these classes contains another qdisc, typically tc-pfifo. When a packet is being enqueued, HTB begins at the root and employs various methods to determine which class should receive the data. Unless there are uncommon configuration settings, the process is relatively straightforward. At each node, it looks for an instruction and then proceeds to the class indicated by that instruction. If the found class is a leaf node (which means it has no children), the packet is enqueued there. If it is not yet a leaf node, the process is repeated, starting from that node.

The test being performed involves limiting the total throughput to 1000Mbit/s and then dividing the available bandwidth between two flows, allocating 80% of the bandwidth to one child and 20% to the other. In listing 6.3, you can see the configuration of the HTB qdisc used for this test.

```
#!/bin/bash

UP_TOTAL_RATE="1000"
MSC1_SP1_RATE="200"
MSC1_SP2_RATE="50"

INTERFACE="ens2f1"

sudo tc qdisc add dev $INTERFACE root handle 1: htb

sudo tc class add dev $INTERFACE parent 1:0 classid 1:12 htb rate ${
    UP_TOTAL_RATE}Mbit ceil ${UP_TOTAL_RATE}Mbit

sudo tc class add dev $INTERFACE parent 1:12 classid 1:10 htb rate ${
    MSC1_SP1_RATE}Mbit ceil ${UP_TOTAL_RATE}Mbit

sudo tc qdisc add dev $INTERFACE parent 1:10 bfifo limit 10000

sudo tc class add dev $INTERFACE parent 1:12 classid 1:11 htb rate ${
    MSC1_SP2_RATE}Mbit ceil ${UP_TOTAL_RATE}Mbit

sudo tc qdisc add dev $INTERFACE parent 1:11 bfifo limit 10000

echo -n "Setting filters... "

```

```
sudo tc filter add dev $INTERFACE parent 1:0 protocol all prio 0 u32
    match ip src 1.1.1.1 match ip dst 2.2.2.1 match ip sport 1000 0
    xffff match ip dport 2000 0xffff match ip protocol 17 0xff flowid
    1:10
sudo tc filter add dev $INTERFACE parent 1:0 protocol all prio 0 u32
    match ip src 1.1.1.1 match ip dst 2.2.2.1 match ip sport 1001 0
    xffff match ip dport 2001 0xffff match ip protocol 17 0xff flowid
    1:11

echo "done."
```

Listing 6.3: Configuration of HTB Qdisc: a maximum total rate is set, and then the bandwidth is divided between the two flows, allocating 80% to one flow and 20% to the other

Regarding the XDP program for implementing the Weighted Fair Queueing algorithm, a HASH map has been introduced. This map uses the session identifier as the key and a struct as the value, containing all the values needed to determine the state of that specific flow.

```
typedef struct _key{
    __be32 saddr;
    __be32 daddr;
    __be16 sport;
    __be16 dport;
    __u8 proto;
} session_id;

struct flow_values {
    __u32 pkts;
    __u64 last_prio;
    __u16 weight;
```

```
13 };
14
15
16 struct {
17     __uint(type, BPF_MAP_TYPE_HASH);
18     __type(key, struct session_id);
19     __type(value, struct flow_values);
20     __uint(max_entries, 16384);
21 } flow_weight SEC(".maps");
```

Listing 6.4: Data structures used in the XDP program to handle Weighted Fair Queueing

When a packet arrives at the input interface, the first step is to determine the flow to which it belongs. Afterward, we retrieve the necessary values from the map. Inside this structure, we find the weight assigned to the flow and the priority at which the last packet belonging to that flow was enqueued. At that point, the priority is calculated by adding a value calculated based on the packet's length and the flow's weight. Starting from the last priority to which a packet from that flow was queued, an amount equal to `(packet_length * flow_weight)` is added. This calculation allows queuing packets belonging to lower-priority flows in higher priority queues in PIFO (thus, they will be served later). The implementation of this algorithm was inspired by that found in this repository [20]. The code for this is shown in listing 6.5.

```
1 SEC("xdp")
2 int xdp_redirect(struct xdp_md *xdp)
3 {
4     int ret;
5     struct flow_values *session;
6     __u64 start, prio;
7     struct session_id {0};
```

```
8
9       new_flow.weight = default_weight;
10
11      __u32 pkt_len = (xdp->data_end - xdp->data) & 0xffff;
12
13      /* Checks to see if the packet is well formatted */
14      /* During the checks, the session ID is extracted */
15
16      session = bpf_map_lookup_elem(&flow_weight, &session_id);
17      if (!session)
18          return XDP_DROP;
19
20      session->pkts++;
21
22      /* Calculating priority */
23      start = bpf_max(time_bytes, flow->last_prio);
24
25      flow->last_prio = start + (pkt_len  * flow->weight);
26
27      prio = start;
28
29      if (bpf_map_update_elem(&flow_weight, &session_id, flow, BPF_ANY)
    )
30          return XDP_DROP;
31
32      ret = bpf_redirect_map(&pifo_map, prio, 0);
33
34      return ret;
35
36  }
```

Listing 6.5: Procedure to follow to decide priority for a packet in weighted fair queueing XDP program

The test results are shown in Figure 6.5. It can be observed that the throughputs in both cases tend to adhere to user-imposed rules, effectively dividing the bandwidth between the two flows.
Although the throughputs are equal, it is crucial to note the difference in resource utilization between the two cases. In both cases, traffic from the two flows is handled by a single core. In the case of the HTB qdisc, this core is occupied at 70%, while in the other case, it is occupied at 7%. This once again underscores the advantage of using the XDP framework, which allows for more available resources.

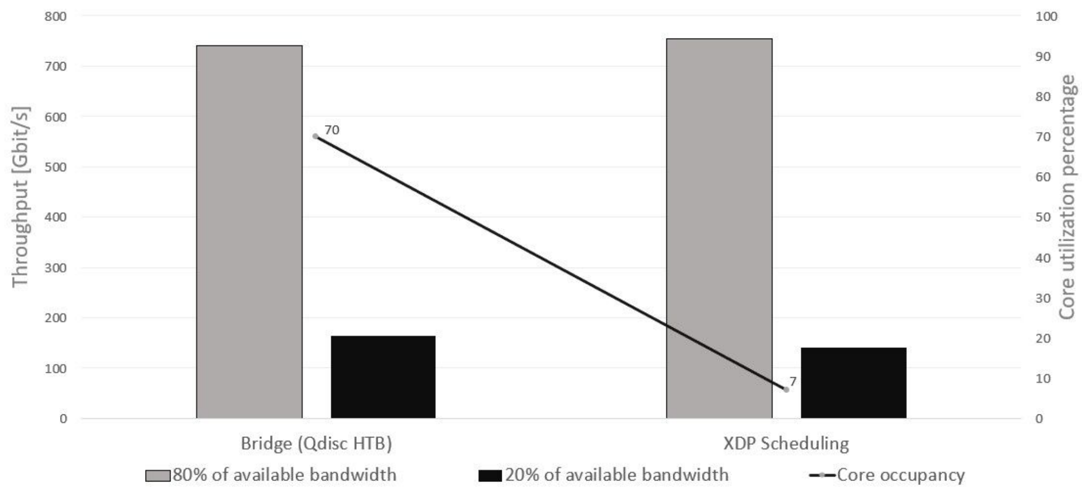

Figure 6.5: Comparison in throughput and core utilization between HTB Qdisc and XDP program - 2 TCP Flows redirected to one core

6.4 PIFO's expressiveness

A mechanism for customizable scheduling must include a data structure that is flexible and highly expressive to cover a wide range of algorithms that can be implemented through it. The PIFO structure and its software optimization, Eiffel, offer rich expressiveness.
Scheduling algorithms commonly used in modern networks involve the presence of

hierarchies among classes. An example is the HTB qdisc, where it is possible to define the root of the qdisc and child classes that are linked to the parent. Each class, in turn, can contain other classes. HTB falls into the category of classful qdiscs.
The patch analyzed in this thesis work involves the construction of class hierarchies (in our case, a hierarchy of PIFO maps). The patch introduces a map type called `BPF_MAP_TYPE_PIFO_XDP`, which can be used to redirect packets and has been used until now. In addition to this, another map type called `BPF_MAP_TYPE_PIFO_GENERIC` is introduced, where any data structure can be enqueued. This map can be used, for example, as the parent of the hierarchy to which two `BPF_MAP_TYPE_PIFO_XDP` maps can be attached as children. The parent map can be used to insert tags (simple integers may suffice) that indicate from which child map to remove the next packet.

The helpers that you can use to redirect data towards generic PIFO are different from the ones that we saw before. The helper `long bpf_map_push_elem(struct bpf_map *map, const void *value, u64 flags)` can be used to send any type of data in the `BPF_MAP_TYPE_PIFO_GENERIC` map, while the `long bpf_map_pop_elem(struct bpf_map *map, void *value)` is used to retrieve data from it. The logic used by the XDP program is shown in Listing 6.6, where the maps used and how packet dequeue is managed from the parent to the leaf map are demonstrated. If the constructed hierarchy has multiple levels, the operation must be repeated recursively until reaching the leaf map from which packet dequeue is to be performed.

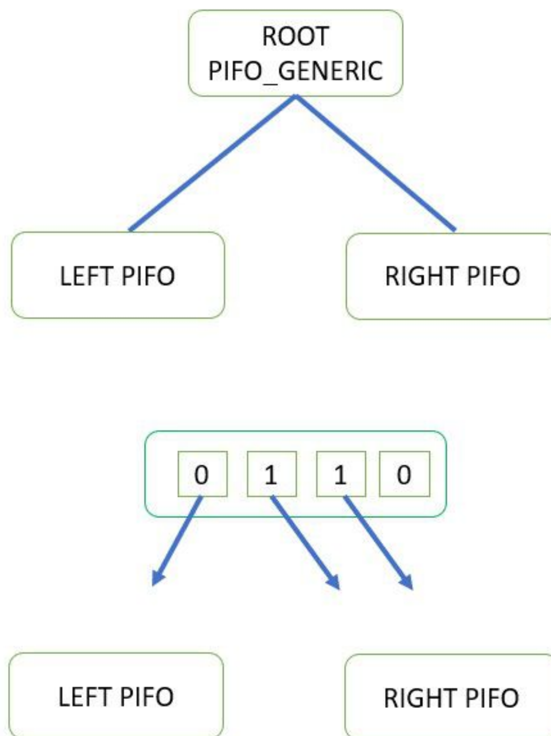

Figure 6.6: Example of building a hierarchy with PIFO maps. In the parent map, tags are inserted to indicate from which map to perform the next dequeue

```
struct {
	__uint(type, BPF_MAP_TYPE_PIFO_GENERIC);
	__uint(key_size, sizeof(__u32));
	__uint(value_size, sizeof(__u32));
	__uint(max_entries, 4096);
	__uint(map_extra, 268435456);
} root_pifo SEC(".maps");

struct {
	__uint(type, BPF_MAP_TYPE_PIFO_XDP);
	__uint(key_size, sizeof(__u32));
	__uint(value_size, sizeof(__u32));
	__uint(max_entries, 4096);
	__uint(map_extra, 268435456);
} left_pifo SEC(".maps");

struct {
	__uint(type, BPF_MAP_TYPE_PIFO_XDP);
	__uint(key_size, sizeof(__u32));
```

```
    __uint(value_size, sizeof(__u32));
    __uint(max_entries, 4096);
    __uint(map_extra, 268435456);
} right_pifo SEC(".maps");

static __always_inline int schedule_packet(){
    __u32 leaf_id;
    __u64 root_prio = 0;
    __u64 leaf_prio = 0;

    if (bpf_map_pop_elem(&root_pifo, &leaf_id))
        return NULL;

    if (leaf_id == LEFT_PIFO){
        pkt = (void *) bpf_packet_dequeue_xdp(&left_pifo, 0, &leaf_prio);
    }else{
        pkt = (void *) bpf_packet_dequeue_xdp(&right_pifo, 0, &leaf_prio);
    }

    if (!pkt)
        return NULL;

    bpf_packet_send(pkt, 9, 0);
    return 0;

}
```

Listing 6.6: Maps used for building a simple hierarchy and a function used for packet dequeue. First, it is determined from which child map the packet should be retrieved, and then the action is performed

Chapter 7

Conclusions and future work

The goal that Tiesse aims to achieve is to obtain a traffic acceleration system through software that can perform functions in the same way that Linux traffic control already does, but with a reduced load on system resources. There are multiple alternative solutions that can be attempted, and one of them is certainly eBPF technology, particularly at the XDP hook point. The use of this technology is becoming increasingly popular and widely shared within the Linux community, and the number of studies and projects based on eBPF is growing year by year. This demonstrates that eBPF is recognized as a promising technology that will likely play a fundamental role in future network traffic developments.

The prototype presented shows excellent performance in terms of throughput and system resource utilization. The prototype presented shows excellent performance in terms of throughput and system resource utilization. Scalability also does not seem to have any issues.

Regarding the analyzed patch, it can be said that the features introduced by the patch could bring significant benefits to the usability and flexibility of the XDP block. Until now, it was not possible to have a structure in XDP to store the packet. The decision about the packet had to be made immediately and could not be deferred to a later time. The slowdown introduced by queuing in vanilla XDP

was demonstrated, along with some potential capabilities that can be exploited. The goal of the patch is to make the XDP block a possible replacement for the Qdisc layer in traffic control. Of course, the features demonstrated in this thesis work cover only a small percentage of the features that a layer managing QoS (Quality of Service) in a network device should have, but it seems to be heading in the right direction.

7.1 Future developments

Of course, the prototype is not yet complete. The services that a network device must perform are numerous and extend beyond those implemented in this thesis work. Managing traffic through the XDP block involves a complete rewrite of network functions performed by the Linux network stack. This includes also firewalling services. In Linux, this service is managed by iptables, which, through various input, output, and forward chains, attempts to block access to specific flows to the system. An iptables system based on eBPF can be more scalable and performant than iptables managed by vanilla Linux [21]. However, the challenge is to create a prototype that queues all of these functions and still manages to be scalable and advantageous compared to vanilla Linux.
Another service that could be managed by the XDP block is load balancing and DDoS attack mitigation. These services could be implemented as an extension of this prototype in possible future thesis work at Tiesse.
Other than that, it is important to see if the patch is merged into the main Linux kernel. At that point it will really be possible to study the definitive API in depth (which may be slightly different from the one analyzed). When it will be present in the main kernel, the goal is to choose some qdiscs of greatest interest and most used within network devices and reproduce their behavior using PIFO maps. It will be necessary to scale the operation of these eBPF-based qdiscs to truly be a

viable alternative to the Qdisc layer.

Another work to keep an eye on is a patch [22] which aims to make qdiscs programmable via eBPF. The two jobs are unlikely to converge since one works in XDP and one in the TC but both use a map to store network packets.

Bibliography

[1] eBPF authors. *what-is-ebpf*. https://ebpf.io/. Accessed: 2023-08-25 (cit. on p. 3).

[2] *Pinning maps*. https://ants-gitlab.inf.um.es/jorgegm/xdp-tutorial/-/tree/344191124593c32497505606075524ed8e5b24df/basic04-pinning-maps. Accessed: 2023-08-25 (cit. on p. 7).

[3] *Linux manual page*. https://man7.org/linux/man-pages/man7/bpf-helpers.7.html. Accessed: 2023-08-25 (cit. on pp. 7, 34).

[4] Toke Høiland-Jørgensen et al. «The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel». In: *Proceedings of the 14th international conference on emerging networking experiments and technologies.* (2018), pp. 54–66 (cit. on p. 9).

[5] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal. «Creating complex network services with ebpf: Experience and lessons learned». In: 2018 (cit. on p. 10).

[6] *Traffic Control*. https://man7.org/linux/man-pages/man8/tc.8.html. Accessed: 2023-08-25 (cit. on p. 13).

[7] *xdp: Add packet queueing and scheduling capabilities*. https://lwn.net/Articles/901046/. Accessed: 2023-09-24 (cit. on p. 14).

[8] Anirudh Sivaraman et al. «Programmable packet scheduling at line rate». In: *Proceedings of the 2016 ACM Conference on Special.* (2016), pp. 44–57 (cit. on p. 15).

[9] Ahmed Saeed et al. «Eiffel: Efficient and flexible software packet scheduling». In: *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019* (2019), pp. 17–31 (cit. on p. 15).

[10] *Traffic Shaping and Traffic Policing*. https://www.cisco.com/c/en/us/support/docs/quality-of-service-qos/qos-policing/19645-policevsshape.html. Accessed: 2023-08-25 (cit. on p. 23).

[11] *bpftool-gen*. https://manpages.ubuntu.com/manpages/focal/man8/bpftool-gen.8.html. Accessed: 2023-08-25 (cit. on p. 28).

[12] *BCC to libbpf conversion guide*. https://nakryiko.com/posts/bcc-to-libbpf-howto-guide/#bpf-skeleton-and-bpf-app-lifecycle. Accessed: 2023-08-25 (cit. on p. 29).

[13] *Introduce to BPF timer*. https://lore.kernel.org/bpf/20210715005417.78572-4-alexei.starovoitov@gmail.com/. Accessed: 2023-08-25 (cit. on p. 40).

[14] *Tips and Tricks for Writing Linux BPF Applications with libbpf*. https://pingcap.medium.com/tips-and-tricks-for-writing-linux-bpf-applications-with-libbpf-404ca94daaee. Accessed: 2023-08-25 (cit. on p. 43).

[15] *BPF support for global data*. https://lore.kernel.org/bpf/20190409212018.32423-1-daniel@iogearbox.net/. Accessed: 2023-09-23 (cit. on p. 44).

[16] *Trex: realistic packet generator*. https://trex-tgn.cisco.com/. Accessed: 2023-09-23 (cit. on p. 47).

[17] *LS1046A Freeway Board*. https://www.nxp.com/design/software/qoriq-developer-resources/ls1046a-freeway-board:FRWY-LS1046A. Accessed: 2023-09-23 (cit. on p. 54).

[18] *PRIO Qdisc*. https://man7.org/linux/man-pages/man8/tc-prio.8.html. Accessed: 2023-09-24 (cit. on p. 60).

[19] *HTB Qdisc*. https://man7.org/linux/man-pages/man8/tc-htb.8.html. Accessed: 2023-09-26 (cit. on p. 64).

[20] *bpf-examples*. https://github.com/freysteinn/bpf-examples/commits/xdp_scheduler_tester. Accessed: 2023-09-29 (cit. on p. 67).

[21] Matteo Bertrone, Sebastiano Miano, Fulvio Risso, and Massimo Tumolo. «Accelerating Linux Security with eBPF iptables». In: *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos* (2018), pp. 108–110 (cit. on p. 74).

[22] *Programmable Qdisc with eBPF*. https://lore.kernel.org/all/20220602041028.95124-1-xiyou.wangcong@gmail.com/. Accessed: 2023-09-26 (cit. on p. 75).