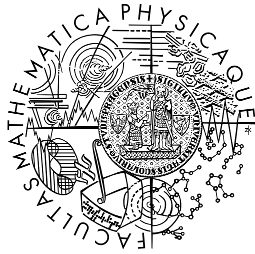

FACULTY OF MATHEMATICS AND PHYSICS
Charles University

MASTER THESIS

Bc. Ondřej Hlavatý

Network Interface Controller Offloading in Linux

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	prof. Ing. Petr Tůma, Dr.
Study programme:	Computer Science
Study branch:	Software Systems

Prague 2018

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date .... signature of the author

Title: Network Interface Controller Offloading in Linux

Author: Bc. Ondřej Hlavatý

Department: Department of Distributed and Dependable Systems

Supervisor: prof. Ing. Petr Tůma, Dr., Department of Distributed and Dependable Systems

Abstract: Modern network interface controllers allow the host to offload packet processing to hardware in order to improve performance. At the present time, the advanced features are utilized in the Linux kernel by offloading the Traffic Control subsystem. Since this subsystem has been designed for a completely different purpose, its usage for hardware offloading is impractical and unreliable. Furthermore, in its current state the subsystem is not capable of utilizing all hardware features, which are often poorly documented.

The presented work adopts a different approach to the problem. Five high-end controllers and their packet-processing pipelines were examined in detail. Accounting for their projected future development, common traits and features were identified. The researched information was used to draft a proposal for a new Linux subsystem, more compatible with hardware offloading than the current solution. The proposed subsystem defines a sufficiently descriptive interface to utilize the majority of hardware-offloaded features while avoiding common problems caused by excessively generalized approach of Traffic Control.

Keywords: hardware offloading network linux traffic control

I would like to thank my supervisor prof. Ing. Petr Tůma, Dr. for guidance. Also, my consultant Jiří Benc and the Red Hat company, for lending me hardware and offering me the topic. Last but not least, my family and friends, for supporting me throughout my studies. Thank you.

Contents

1 Introduction **3**
1.1 Scope 4
1.2 Linux Network Stack 5

2 Hardware Offloading in General **7**
2.1 Checksum Offload 7
2.2 TCP Segmentation Offload 8
2.3 UDP Fragmentation Offload 9
2.4 TCP Offload Engine 10
2.5 Multiple Queues 10
2.5.1 Multiple Send Queues 10
2.5.2 Multiple Receive Queues 11
2.6 Flow Offload 13
2.6.1 VLAN and Tunnel Offload 14
2.6.2 Access Control Lists 15
2.6.3 Match-Action Offload 15

3 Selected Controller Capabilities **17**
3.1 Intel 82599 10 Gigabit Ethernet Controller 17
3.1.1 Checksum Offload 17
3.1.2 Segmentation Offload 18
3.1.3 Multiple Receive Queues 18
3.1.4 Multiple Transmit Queues 20
3.1.5 Other offloads 20
3.2 Intel Ethernet Controller XL710 20
3.2.1 Stateless Offloads 21
3.2.2 Multiple Queues 21
3.3 Mellanox ConnectX-4 23
3.3.1 Switch Layout 24
3.3.2 Pipeline 24
3.3.3 Stateless Offloads 25
3.4 Chelsio Terminator 6 26
3.4.1 Stateless Offloads 26
3.4.2 Match-Action Offload 26
3.4.3 Other Offloads 27
3.5 Netronome NFP-6000 28

4 Hardware Offloading in Linux **31**
4.1 Ethtool 31
4.2 Features 31
4.2.1 Checksum Offload 33
4.2.2 Segmentation Offload 35
4.2.3 TCP Offload Engine 37
4.3 Multiple Queues 37
4.3.1 Receive-Side Scaling 38

4.3.2 Receive Packet Steering 38
4.3.3 Receive Flow Steering 38
4.3.4 Ethtool Network Flow Classification 39
4.3.5 Transmit Packet Steering 40
4.4 Express Data Path 40
4.5 Traffic Control 41
4.5.1 Queue Disciplines 41
4.5.2 Filters 43
4.5.3 Actions 43
4.5.4 Actions on Egress 44
4.5.5 Modularity 45
4.5.6 Offloading 45
4.5.7 Shared Blocks 46

5 Proposed Subsystem **49**
5.1 The Big Picture 50
5.1.1 Matching Modes 51
5.1.2 Actions 52
5.1.3 Offloading 53
5.2 Acting as an OpenFlow Backend 54
5.3 Compatibility 54
5.4 Configurable Parser 55
5.5 Introspection 56
5.6 Acting as a P4 Backend 57
5.7 Simplifying Overlay 57
5.8 Performance 58

Conclusion **59**

Bibliography **60**

List of Figures **63**

Acronyms **65**

A The Demonstrator **67**
A.1 Usage 67
A.2 Implementation Overview 67
A.2.1 Tests 68
A.2.2 Match-Action Tables 69

1. Introduction

The networking technology development is a never-ending race towards wider bandwidth, lower latencies and higher rates of processed packets. Where general-purpose CPUs stay behind, dedicated hardware can increase network performance. Modern Network Interface Controllers (NICs), in addition to connecting the host computer to the network, also have advanced features that assist with processing packets. When hardware is used to perform a part of a job originally done in software the technique is usually called *hardware offloading*.

Recently, high-end controllers learned how to classify and modify packets, for example, to drop packets with certain properties or to automatically extract an inner packet encapsulated in a tunneling protocol. These features are useful (not only) in environments where virtual machines running on shared hosts communicate via isolated virtual networks that span over shared physical wires and devices. The less time the host spends on preprocessing the network traffic for the virtual machines, the more is left for doing the fruitful work.

On the opposite end of the network stack are applications that process packets while still being a part of the network function. For example Open vSwitch implements a full-featured software switch. There is a considerable interest in offloading its work to hardware to increase speed and lower resource consumption.

To use the advanced processing capabilities of the controllers, the configured policy must find its way from the userspace to the hardware. Currently, the kernel is often bypassed with solutions like Data Plane Development Kit (DPDK), allowing an application to configure the controller directly from userspace, using more features of the card to process packets. However, the software is highly specialized for this purpose and generally cannot be combined with features the Linux network stack provides.

In the Linux kernel, there are several mechanisms that can be used for generic in-kernel packet processing – among others Netfilter, TC, XDP. Unfortunately, none of them really fits to be offloaded using the packet modification capabilities of the controllers.

New subsystems could be created in the Linux kernel to support specific features of the individual controllers. However, the Linux kernel philosophy is to abstract away from the hardware, so the subsystems created would have to work even without the specific hardware installed. Therefore, solutions which would support devices from a single vendor only are likely to be rejected by the community.

Because the hardware release cycle is long compared to the speed of evolution of modern networking, the features that vendors put in their controllers are getting more and more flexible. Between designing the controller and starting to sell a finished adapter, new protocols are being invented. For the hardware, being flexible is the only way to keep up with the software.

The flexibility of the controllers can be utilized to reduce differences between the individual controllers, allowing to create a subsystem which would be offloadable by multiple controllers. The main goal of the thesis is to design such a subsystem. The subsystem should provide the glue between userspace doing packet processing and the drivers of the network controllers, creating a generic

platform for packet processing offloads. Ideally, any configuration should work independently of the hardware installed, while allowing software to offload as much work as possible to the hardware.

To achieve this goal, we selected five recent high-performance controllers and decided to examine their capabilities in detail. As with the simpler offload techniques, there is no literature which would contain the needed information with the right amount of detail. There are advertisements and marketing articles, which present rather vague terms and keywords, but usually do not give any idea of how the controller works. Some controllers have manuals for proprietary drivers from which the range of available features can be deduced, but we cannot tell apart the work done by the driver and the controller. Then there is the source code of the Linux kernel and the DPDK that can give us a very good understanding of the features which are utilized, but only after decoding the big codebase of the relevant drivers. Finally, there are public datasheets and manuals for some controllers that contain all the information needed, scattered in hundreds of pages with additional information which is not relevant.

One of the most painful problems of Linux is that its documentation cannot keep up with the immense speed of development. Usually, the initial idea is documented, presented on conferences and so on, but subsequent changes do not update the overall picture presented in the documentation. Therefore, the current state of a feature is hard to understand if one does not follow its development from the beginning.

The majority of information provided in this thesis is gathered from the source code directly or assembled from little pieces found in the kernel documentation and commit messages, providing the complete image of the current state of hardware offloading. It is, to our knowledge, the only document of its kind.

In summary, this thesis contributes:

- The review of the hardware offloading techniques in Chapter 2.
- The review of the features of the selected controllers in Chapter 3.
- The overview of the Linux kernel with respect to the network hardware offloading in Chapter 4.
- The proposal of the new subsystem in Chapter 5.

1.1 Scope

When talking about networking, we mostly limit ourselves to IP over Ethernet. We are not fond of supporting protocol ossification, but IP and Ethernet are arguably the most widespread technologies in the computer networking. As for the network layer protocol, watching IPv6 having hard time replacing IPv4, it is hard to imagine a completely different protocol taking over. For the Ethernet, the situation is curious. A lot of different communication standards over different media share the common Ethernet marketing label. It is the presence of the common paradigms that allows the network controllers to support multiple Ethernet standards, making a gradual transition to newer standards possible.

For high-performance networking, other communication technologies are available (e.g. InfiniBand), but their support and adoption by the operating systems is far from that of Ethernet and IP.

At the time of writing the thesis, the most recent released version of Linux kernel was 4.16 [30]. All the information about the kernel is based on this version. As the topic is still hot, we also used the David Miller's net-next tree [22] with the most recent updates for the networking subsystem to learn more about the controllers. However, we do not refer to commits from there, due to their experimental nature.

1.2 Linux Network Stack

As the thesis is requires the reader to orient briefly in the Linux networking stack, a condensed and very simplified overview follows. We skip a lot of detail and intermediate packet processing and focus on parts which are important to understanding the rest of the thesis. A more comprehensive description can be found in [28], but the only literature that is always up-to-date is the source code itself.

Let us explore the life of a datagram being transmitted using UDP over IPv4 between two applications that run on Linux hosts. First, we will look at the *egress* direction (sending the packet from the host to the network), then the *ingress* direction (receiving a packet from the network). Suppose that the datagram is small enough to be delivered in one IP fragment and let us ignore all the errors that might happen.

When an application wants to communicate via the network in Unix-like operating systems, it opens a *socket*. The socket is an entity in kernel memory that can be controlled by the application using a handle (a file descriptor). The socket can be created using a system call of the same name. In our case, the socket is created with the `AF_INET` address family (IPv4), the `SOCK_DGRAM` socket type to communicate using datagrams and the `IPPROTO_UDP` protocol to encapsulate data in the UDP.

Once created, the corresponding file descriptor is used as an argument to subsequent system calls, controlling the entity in the kernel. For the sender, no further setup is necessary, the socket is ready to send datagrams right away. The sending application prepares the data in a memory buffer, and requests them to be sent by the `sendto` or `sendmsg` system calls. (The `write` or `send` system calls can be used as well, but the socket must be configured with the intended recipient first.)

No matter which system call was used, it is handled by calling a `sendmsg` protocol callback – in our case, the `udp_sendmsg` function. To move the packet-related data around the kernel, an `sk_buff` structure is created. This structure represents a data buffer in the networking subsystem and is used almost everywhere. Its lifetime is dynamic, thus reference counting is employed.

Once the `sk_buff` is created, it is filled with already known metadata and packet headers are constructed. An unexpected fact is that the IP header is filled earlier than the UDP header. As UDP is closely tied to IP, the layered network model is not followed strictly in Linux.

The next important decision to be made is routing the packet. Routing (among other things) selects the port that will be used to push the packet out from the host. Ports are represented by the `net_device` structures in Linux. It

is common for high-performance NICs to present multiple ports to the system, multiple net_devices may then correspond to a single physical controller.

As there might be multiple applications trying to send data from the selected device, the packets are not given directly to the driver. Instead, they are temporarily stored in queues in the Traffic Control (TC) subsystem. A detailed look at the subsystem is provided in Section 4.5.

The NIC usually utilizes circular queues to communicate with the host. Packets are dequeued by the controller at its convenience. When there is an empty slot for a new packet (and such a packet is available), it is dequeued from TC and handed out to the driver.

The net_device structure, among many things, carries a pointer to a static instance of net_device_ops structure. This structure holds callbacks that implement the network device interface for the rest of the system. One of the most important callbacks is ndo_start_xmit, which is called to transmit a packet.

The driver usually needs to fill some descriptor structure for the packet, configuring the processing that will happen in the hardware. An obvious part of the descriptor is the memory location of the buffer where the packet is stored. Once the descriptor is ready, its virtual ownership is transferred to the controller.

But the processing for the sending host is not over yet. The memory used by the packet must not be deallocated, because it is potentially accessed by the controller in the background. Therefore, the driver still holds one reference for the sk_buff and drops it only after the respective slot in the hardware queue is marked as empty.

Before the packet reaches the kernel on the other host, the receiving application must be prepared to receive it. If it was not, the kernel would drop the packet as not wanted. The application does so by opening a socket and *binding* it. When the socket is bound to an address and port, the kernel notes that packets sent to this destination should be delivered to this particular socket.

The ingress path of a packet is a bit simpler. First, the host needs to prepare memory for the received packets in advance. It allocates free pages and enqueues the receive descriptors, similarly to what it does when sending packets.

Once the controller decodes a network frame from the medium, it copies it to the prepared memory. As there is usually some processing in the controller itself, some metadata about the packet is already known. Some of the metadata is stored inside the descriptor, to potentially speed up the processing on the host.

Next, the controller marks the slot in the receive queue as ready. The driver picks it up and creates an sk_buff structure for the received packet. It can use some metadata from the descriptor to pre-fill fields of the sk_buff. Then the driver calls netif_receive_skb to give the received packet to the network stack.

First, the packet is parsed to identify some header fields, up to the transport layers. This is needed to support some early optimizations, which are further discussed in Section 4.3.2. In contrast to the egress path, there is no buffering of packets in TC, all packets are delivered immediately. In case of IPv4, the delivery is realized by calling the ip_rcv function.

Similarly to egress, the routing tables are consulted to decide whether the traffic is local. If so, the UDP handler is called, which enqueues the packet in the socket queue. There, the packet waits until it is picked up by the application calling the recvmsg, recvfrom, recv, or read system calls.

2. Hardware Offloading in General

In this chapter we would like to introduce the current techniques used to reduce the amount of work that the network stack has to do in software (by the host CPU) by doing it in the NIC instead. The simplest techniques are implemented by virtually every NIC currently on the market, the interfaces are stable and well-supported. On the other end of the spectrum there are techniques which are specific to high-performance cards and are under rapid development at the time of writing.

The information in this chapter is based on capabilities of several high-end controllers (further discussed in Chapter 3), Microsoft Network Driver Interface Specification (NDIS) [21], or available documentation for the Linux kernel.

All of the information given in this chapter is public. However, there is no comprehensive overview of available techniques. Individual offloads are given business names and the implementation is buried deep inside datasheets. Available documentation for drivers and operating systems focuses on how to control the mechanisms and when it is good to use them, but usually do not cover the principles behind. Therefore, this chapter aims to be an introduction to modern networking world.

It is worth noting that every vendor of a network controller uses their specific terminology. The implementation details might differ as well, but the high-level principles described in this chapter are implemented by multiple NIC vendors.

An important aspect of an offload is that the functionality provided by the hardware is not required for packets to be processed. The functionality is strictly opt-in and only if the driver of the device is capable of doing so, it can enable it and benefit from skipping some steps in the network packet processing on the host.

At first, we would like to mention two techniques that can be classified as offloads, though this requires a rather loose interpretation of the term. First is the scatter-gather capability of the controller, which frees the host from assembling multiple buffers into one. Second is the interrupt moderation, which reduces the overhead of switching contexts and allows to batch-process multiple packets. Both of these techniques save some CPU cycles to process a packet, but they do not depend on the packet content.

2.1 Checksum Offload

A common approach to ensure consistency of data transmitted over a network involves checksums. Those are values that are usually inexpensive to compute, yet provide a good level of reliability in indicating data corruption.

At the link layer the NIC usually computes the checksum itself, because it is well aware of the protocol being used. In the case of Ethernet the checksum is called *Frame Check Sequence* and is transmitted at the end of the frame, after the payload [14]. Therefore, the NIC is able to compute the intermediate value of the checksum while sending and finish the frame by the final checksum. Similarly,

on the ingress path, after a frame is received completely (with the checksum included), the value should be zero – otherwise the frame is dropped.

Moving up to the network layer, IPv6 does not embody a checksum at all [8]. The IPv4 header contains a one's complement checksum of the header itself [27]. The maximum size of the header is 60 bytes, therefore the checksum is not expensive to compute in software. Still, many controllers are capable of computing the IPv4 header checksum before transmission.

Matters get complicated at the transport layer. A handful of protocols uses 16-bit one's complement checksum of the whole packet – not only TCP and UDP, the same approach is used by DCCP or GRE, for example. The offset at which the checksum field is placed in the header varies with the protocol.

Computing the checksum of the whole transport-layer packet is expensive for the operating system, because the packet payload can be large. Also, as it is not accessed otherwise, it is poorly cached. Therefore, the kernel must read the whole payload from memory just to compute its checksum.

Both receive and send offloading capabilities for the transport layer checksums are commonly offered by NICs. On the receiving side, a NIC can either validate the packet directly or just calculate the checksum including the checksum field. The second way is preferred because of its flexibility, as we will see in detail in Section 4.2.1.

Due to the algebraic properties of the checksum, it is not necessary to know where the checksum field is to verify the received packet, making the receive checksum offload both simple and protocol-agnostic. However, the transport-layer packet must be received completely, in other words, the IP packet must not be fragmented.

When sending a packet, older NICs might be able to only check for the presence of a known transport protocol and compute the checksum for them. Recent controllers have the ability to compute checksum of any suffix of the packet and write it to an arbitrary offset, allowing the system software to offload compatible protocols in a generic way.

Both TCP and UDP compute the checksum not only from the packet header and payload, but also from a *Pseudo Header* which is composed of the source and destination addresses. These are not duplicated in the transport layer PDU, and because of that the device might require the system software to precompute the checksum of the pseudo header, exploiting the associativity and commutativity of the checksum algorithm. An example of this requirement can be found in the Intel XL710 Controller ([16], Section 8.4.4.3.2). When a packet is received, its computed checksum (including the checksum field) should be equal to a complement of the pseudo header checksum.

For different checksumming algorithms, alternatives are also often provided. For example, the Intel 82599 Controller can offload the CRC of SCTP ([15], Section 7.2.5.3)

2.2 TCP Segmentation Offload

As TCP is offering an interface of a stream pipe, it includes a rather complicated mechanism which creates the illusion of the pipe on top of a network that is only able to transmit individual packets of bounded length. When chunks of data are

inserted into the pipe, they are split into *segments*. Segments are labeled with sequential numbers and sent individually. The sequential numbering is used to ensure that no segments are lost and data is delivered (passed to the application layer) in the correct order. However, the size of the chunks is not defined. Taken to the extreme, TCP could deliver individual bytes.

Obviously, a lot of overhead can be mitigated by handling data in the biggest chunks possible, traversing the entire stack the least number of times. Speaking about software, the network stack usually performs the segmentation at the latest possible moment, performing as much processing as possible in batches. When receiving, the stack might try to coalesce segments of a single TCP stream before delivering them.

This effort can be further supported by the hardware. A NIC can offer an interface which allows to enqueue TCP packets that are larger than the link MTU. Before transmitting them on the wire, the controller splits the packet into multiple segments by itself. This way, the network stack is no longer bound to the link MTU and can work with coarser chunks. This feature is often referred to as Large Send Offload (LSO).

As this offload needs to have insight up to the TCP header, only a limited set of header combinations is typically supported. Apart from plain TCP over IP, more advanced NICs can support TCP encapsulated in multiple tunnel types.

The receiving counterpart of LSO is usually called Large Receive Offload (LRO) and does the expected opposite. Prior to enqueueing the packet into the receive queue of the operating system, it tries to coalesce multiple received segments belonging to one TCP stream into one super-packet. It is important to make this feature optional, because it would violate the responsibilities of a network when the packet is bridged or routed in the software.

Careful reader might notice that both directions of TCP segmentation offload depend on checksum offload, as the newly-forged packets containing segments need to have their checksum computed by the device on all layers. On the ingress path, the to-be-forgotten packets carrying segments need to be verified prior to coalescing, and the coalesced packet either needs to have a valid checksum or it must be marked in order to skip verification by the software stack.

The checksums are the key issue the controller has to solve to support segmentation offloads over tunnelling protocols, as multiple checksums might need to be computed along the way to the inner TCP packet, while it is the TCP packet payload being segmented. Therefore, not only the TCP checksum, but also the outer UDP checksum may change for e.g. VXLAN.

As a side note, TSO can increase performance in virtualized environments, as it increases effective MTU of the virtual links without breaking isolation.

2.3 UDP Fragmentation Offload

A similar but much simpler technique exists for UDP datagrams. The length of one UDP datagram is limited to 64 KB, larger than MTU of common links. UDP itself does not define any concept of fragmentation, instead it utilizes IP fragmentation to deliver payloads larger than the MTU.

Controllers might be capable of performing IP packet fragmentation on the chip. In contrast to TCP segmentation offload, IP fragmentation can be done

without touching the transport layer PDU, thus only the IPv4 header checksum must be updated.

The offload techniques described so far are often referred to as *Stateless Offloads* despite the fact that e.g. LRO needs to maintain state. Because of its convenience, we will use this terminology as well.

2.4 TCP Offload Engine

Some NICs are equipped with a full TCP stack Offload Engine (TOE). Using this engine, the host leaves TCP/IP stack processing completely on the NIC. There are two main approaches. In the less invasive one, the software stack initiates the connection using regular mechanisms, then hands the stream over to the NIC. From that moment, the NIC offers a complete stream interface, and handles all the TCP-related work – segmentation, congestion control, retransmissions and so on.

The second and even more intrusive approach leaves the TCP connection handling on the hardware from the very beginning. Essentially, the TOE driver replaces the TCP stack completely.

While TOE might seem superior to partial offloads, it brings a lot of controversy. Processing TCP is not a simple task at all, and complex code cannot avoid bugs and security flaws. Updating an operating system is a simple task compared to updating a firmware in a controller. Furthermore, the operating system cannot control the extent of features provided by the TOE, and users would probably be confused by missing features like firewall or QoS. This holds even more so when we consider that these features are still configurable, but have no effect on the offloaded TCP stack, because the respective packets bypass them.

2.5 Multiple Queues

A lot of opportunities to improve performance arise from creating multiple queues for the communication between the host and the NIC. The maximum number of queues in question is specific to the controller and depends on the associated purpose. As we will see, multiple queues can help in more cases than only on multi-processor systems.

2.5.1 Multiple Send Queues

Let us start with increasing the number of egress queues per port. To handle multiple queues, the controller must multiplex them on the wire. The algorithm for multiplexing packets from multiple queues in fact performs packet scheduling.

With the knowledge of the scheduling algorithm, the host can offload scheduling to the NIC by selecting the proper queue for every packet. This requires both the algorithm and the number of queues to be compatible with the desired behavior.

We will show two examples of how multiple queues can be used. In the first one, assume that the scheduling algorithm is round-robin, thus every queue is treated equally (with respect to the number of packets, not bytes sent). In

a sense, the host can treat every queue as a separate interface to a shared medium. On a multiprocessor system, it is possible to dedicate one queue to every core, offloading the synchronization between the cores to the NIC. The downside of this approach is that the bandwidth is not equally and deterministically divided between processes, instead it is affected by the number of threads and thread scheduling.

In the other example, consider a strict-priority scheduling algorithm. In this model, the host can differentiate several traffic classes (e.g. prioritize interactive traffic before bulk traffic) and distribute them among egress queues appropriately. This technique can cut a bit of latency introduced by buffering in the hardware queues of the device. Some scheduling still needs to be done in the software, as there can be multiple sources of packets which need to be multiplexed into one queue.

Scheduling offload is very hard to adopt, because subtle differences between scheduling algorithms of the host and NIC can violate the intent of the system administrator. The NIC driver must make sure that the current scheduling algorithm is compatible with what the NIC employs.

Recent controllers offer some flexibility in terms of configuration of the scheduling algorithm. For example, the Intel XL710 controller [16] defines queue sets, which can be arranged to a tree. The leaf nodes are queue sets, the inner nodes select its children in a configurable combination of weighted strict priority and weighted round robin order. The bandwidth is distributed equally among the queues in a set.

2.5.2 Multiple Receive Queues

Multiple ingress queues can be utilized quite easily. The controller has to distribute the packets received from the wire to the queues and the selection of algorithm the to do so creates space for offloads.

Receive-Side Scaling

To take advantage of multiple processors available for processing network traffic, a simple mechanism called *Receive-Side Scaling* was specified by Microsoft in the Network Driver Interface Specification [21].

The idea is that similarly processed packets should be handled by the same CPU to maximize the benefits of caching. Also, packets from a single flow are likely to be processed similarly. Therefore, an RSS-enabled NIC extracts the source and destination addresses, possibly also the TCP/UDP source and destination ports, and calculates a hash function of those. The lower bits of the result are used to select the target queue. Queues are then uniformly distributed among CPUs and the interrupt affinity is set accordingly.

One might dispute RSS being considered an offload, as it does not have a software predecessor. We can take RSS as a reason why multiple receive queues exist and as an inspiration for all other multi-queue offloads.

Differentiated Services

Instead of merely distributing packets in a stochastic manner, the queues can be assigned a specific purpose. Every IP packet carries a priority field, which can be used to assign a traffic class to every packet. The field is called Type Of Service (TOS) in IPv4 and Traffic Class in IPv6. Traffic classes can then be mapped to receive queues by the NIC. Knowledge of the priority mapping then allows the software stack to handle certain traffic classes with higher priority than the others.

Differentiating traffic classes as soon as possible is important under heavy load. For example, TCP avoids congestion by lowering the transmission rate when a segment is lost. This means that the rate is only lowered when some network element decides to drop a packet. Unless some more sophisticated algorithm to drop packets earlier is employed, packets are dropped only when a queue is overfilled. That results in queues being kept rather full, introducing delays for high-priority traffic as well.

Advanced Classification

The NIC can select the target receive queue based on a more complex algorithm, considering not only the TOS field in the IP header, but also other fields. Usually, the set of fields the controller is capable of matching on is limited. The variants might be mutually exclusive or fixed in linear order of execution. Popular variants include:

- Ethertype
- Source or destination MAC address
- Transport layer 5-tuple (protocol, source and destination address, source and destination port)
- VLAN ID or other tunneling header fields

In addition to the predefined header fields, the NIC can offer a way to define new header fields. The number of configurable fields tends to be very low. On the other hand, several switch ASICs are equipped with packet parsers fully programmable by firmware updates, and therefore we can expect similar flexibility in NIC controllers in a foreseeable future.

Usually, the matching rules are arranged into tables. There can be either a fixed pipeline of tables, or a table hierarchy might be defined at runtime, or a combination of both. Every rule carries a queue number, which is used whenever a packet matches the rule.

Multiple modes of matching on a field can be distinguished. First, the field might be examined for an exact value. Or, every rule can contain a *value* and a emphmask. Rarely, the fields can be compared with ranges or a longest-prefix match can be selected.

In the exact-match mode, a controller can use a special type of memory called Content Addressable Memory (CAM) – a memory where lookup of a row with a given value is done on all rows in parallel. Or, it can use hash tables and store the tables in RAM, but that requires collision handling.

The mask-value rules are used frequently in the networking world, e.g. for routing. For every rule individually, the field value is first masked by the mask

and then compared with the expected value. In hardware, this operation can be realized on all rules in parallel by using a Ternary Content Addressable Memory (TCAM) – a memory addressable by keys in which the rules can ignore individual bits. However, this requires the memory to store at least three states per bit in rule. Therefore, the flexibility is paid for with much higher number of transistors for the same size of the table. Due to their construction, TCAMs also consume a lot of power and take up a lot of space on the chip. Therefore, tables placed in TCAM are usually a lot more constrained than tables in CAM or RAM – in both the total size of the used fields and the number of rows.

Range matching can be implemented with a TCAM extended for this purpose or mapped to regular TCAM matches. Such a mapping can consume as much as $2 \cdot \lceil \log_2(B - A) \rceil$ TCAM rules to represent $x \in (A, B)$ rule. Longest prefix matching is usually implemented by TCAMs with priority such that the lookup result is always the first matching rule. Then, ordering prefixes from the longest to the shortest makes TCAM perform a longest-prefix match.

2.6 Flow Offload

In the last few years, controllers of the network interfaces started to resemble switch controllers. It might seem strange at first, as NICs have usually very few external ports compared to switches. However, modern high-performance NICs usually feature Single Root IO Virtualization (SR-IOV), which presents the controller as multiple PCIe devices to the host. Usually, the first device is called *physical* function, the others *virtual* functions. The virtual functions can be handed out to virtual machines, removing the overhead of communication through the hypervisor. Usually, the virtual functions can be restricted or partially configured via the physical function, supporting virtualization deployments.

The SR-IOV can be seen as multiple virtual ports going out from the NIC, naturally creating the need for switching. The usual scheme is that the adapter external ports as well as the physical and virtual functions are connected as ports to an inner switch. The need for switching becomes even more apparent with multi-host network adapters, such as the Mellanox ConnectX-5 [19] based adapters.

The existence of an inner switch is not hidden by the controller. Inner switches are usually not that performant and flexible as fully-fledged switch controllers, but we can already observe inner switches adopting useful features and complexity of standalone switches.

When it comes to virtualization, a Software Defined Networking (SDN) is usually deployed. The “software defined” aspect means that the physical topology is hidden, creating a virtual overlay network over the physical one. To give an example, consider a data center running virtual machines, that are migrated to balance load. Traditional network elements would react too slowly to enable this scenario, so an externally configurable, more flexible switch is used instead. SDN-configured switches do not try to find paths by themselves, they need to be programmed by a standalone SDN controller explicitly. The controller reactively plans paths for packet flows and configures the network accordingly, replacing the distributed switching algorithm with a centralized one. The communication

between the switch and the controller is realized by a protocol designed for this purpose – for example the OpenFlow protocol [25].

To perform forwarding in an SDN environments, switches feature a flow table. When an unknown packet is received, it is forwarded to the controller. It then identifies the flow and installs a rule into the switch flow table. The next time a packet from this flow is received, it is forwarded without controller intervention. As the memory of the switch is limited, unused flow table entries are evicted.

This model is commonly described as a layered one, the controller being a control plane and the forwarder being a data plane. While the control plane focuses on flexibility and features, the data plane focuses on performance. This separation can be observed in many instances in the networking world.

So far, the inner switches in NICs do not have the required configurability (e.g. they cannot communicate with the controller using the OpenFlow protocol) to fully support SDN. Instead, software switches are being deployed. This opens an opportunity for offloading, which is slowly being implemented by the NICs. Controllers with SR-IOV have the ability to offload the virtual function selection.

To a person with theoretical education in networking, switching is a matter of decision based on the destination MAC address, and all of this might seem as an over-engineered solution. Nevertheless the destination MAC address is no longer the only field used for selecting the destination port. Usually, the same classification engine used for steering packets among receive queues is used for the target port selection. Rarely, as the receive queue is bound to one port, both switching and receive queue selection can happen at the same time.

2.6.1 VLAN and Tunnel Offload

Another new responsibility of switches is tunnel handling. For several years, VLAN-handling features can be found even in SOHO switches. The switch can transparently emulate multiple networks by adding and removing VLAN tags on the client ports. To improve virtual network isolation, the port traffic can be usually filtered based on the VLAN tags. Similar features with tunneling protocols like VXLAN or GRE can be found in high-end switches.

When a host runs multiple virtual machines, it probably needs to be connected to multiple virtual networks. In other words, the tunnel processing must be moved from the standalone switch to the switch inside the host – either the software one, or the inner switch inside a NIC.

Presence of tunnelling prevents the flow offloading described in Section 2.6, because the virtual machine would see packets wrapped in the tunnel header. Therefore, filtering and encapsulation/decapsulation of tunnels must be done as well to enable flow offloading in these scenarios.

The problem of tunnel offloading is its future compatibility and flexibility. It takes a lot of time since the controller is designed until it is used in a network adapter available on the market. Furthermore, the card is expected to last a few years. On the other hand, new tunnelling protocols are still being developed (e.g. the Geneve protocol, first drafted in 2014, is still not standardized but is in active development [12]). The functionality of tunnel engines can be superseded by programmable actions described in Section 2.6.3.

2.6.2 Access Control Lists

Together with switching and/or advanced classification, access control can be implemented in hardware. With all the classification apparatus, dropping matched flows is just a simple extension to switching or queue selection.

Let us show one example where offloading the ACLs can play a significant role – a DoS protection. Suppose a server providing a service, which is under DoS attack. Incoming requests undergo accounting that can consider arbitrary properties of the request (source address, subnet, or even some domain-specific properties like the API key being used). Flows exceeding the configured rate are considered malicious and a rule dropping the packets of this particular flow is installed to the hardware. From now on, the traffic from the attackers can no longer disrupt the service, because the packets are dropped before reaching the first software component.

This scenario could be handled even better by dropping the malicious flows earlier, for example on the boundary router. But the flexibility and cost-effectivity of the described solution is a thing to consider.

2.6.3 Match-Action Offload

All the advanced classification offloads described so far can be further generalized by moving closer to the hardware implementation.

A packet processor usually features a pipeline that can be described as follows. When a packet arrives, it is parsed by a parser. Individual header fields are extracted to a metadata following the packet through the pipeline. Then, a series of match-action steps is performed. In every step, the packet metadata is used to search for a matching entry a table, using match modes previously described in Section 2.5.2. From there, a *match index* is obtained and used to lookup an entry in an action table, which contains a chain of actions for every match index possible. Actions are usually operations on the metadata (set field, copy field value to another field, etc.) or the packet as a whole (drop, pop header). The action results might be reflected in the packet data directly, or just in the metadata carried around. At the end of the match-action pipeline, there is a deparser, which applies the changes in metadata back to the packet to reflect all actions. The last step is necessary to avoid touching the packet data directly from the pipeline, which is complicated.

Notably, this pipeline architecture is well reflected by the OpenFlow protocol, which operates on tables, header fields and actions, and assumes that the packet processing pipeline is similar to above.

In the most common case the processing pipeline is fixed. It might be linear or have branches, but usually cannot contain cycles. Every table in a fixed pipeline has a predefined set of headers matched (and the mode of matching as described in Section 2.5.2), and a set of actions allowed.

For example, the switching decision of an ordinary switch can be described as a table performing a lookup on the destination MAC address. If the address is known, the specified egress port is used. If not, the packet is mirrored to all ports as per the default action.

As another example, we can take the VLAN engine of a simple switch. This step is performed in the egress pipeline of a port (specifically, after the forwarding

database is consulted). The table matches on the VLAN ID. When the engine is turned off, the table is empty and the default action is to pass the packet without modifications. If the engine is turned on for the port, the default action is to drop the packet. When the port is connected to a trunk link, the table matches VLAN IDs of the networks the trunk link belongs to, and the action for all of them is to pass without modifications. If, on the other hand, the link is connected to a host unaware of the virtual networks, the only entry in the table matches the VLAN the host is connected to and the action is to pop the VLAN header. Remember that the default action remains to drop the packet, so the engine performs filtering in the same step as well.

Some controllers have the pipeline at least partially programmable. The software can define a new graph of tables, which is inserted into the pipeline. Those tables can match on an arbitrary subset of header fields and can be used to offload almost any real-world packet processing scenario of known protocols.

The programmability of the pipeline alone is not sufficient to implement protocol-agnostic packet processing. For that to be possible, the parser needs to extract fields from the unknown protocol to match on them. The OpenFlow protocol (in version 1.5) defines a set of matchable fields, which cannot be extended. That limits its usage for switches with programmable pipelines [25].

As stated earlier, there are several switch controllers on the market that are equipped with a parser programmable by firmware updates. Such a controller can be used to implement the processing of protocols that are invented later than the switch, or application-specific protocols in general.

And how does that relate to NICs? We have seen many cases of NICs adopting features from switch drivers, and we expect that this is going to be another one. Supporting this claim is the Mellanox ConnectX-5 controller, which already supports the creation of a graph of tables which do match-action processing of the flow (see Section 3.3.2).

3. Selected Controller Capabilities

This chapter documents the features of contemporary NICs with respect to offloading. We have chosen the following controllers for thorough analysis:

- Intel 82599 10 Gigabit Ethernet Controller
- Intel Ethernet Controller XL710
- Mellanox ConnectX-4
- Chelsio Terminator 6
- Netronome NFP-6000

The selection was influenced by the support for match-action offloading, which is the primary focus of the thesis. We have chosen the most recent controllers supported by the Linux kernel. The order is arbitrary, though we preferred controllers with publicly available specifications, as we know more about them.

3.1 Intel 82599 10 Gigabit Ethernet Controller

This controller was released in mid-2016 and since then, it was embedded into many adapters from different vendors – Intel, HP, Dell, and others. The controller supports Ethernet speeds of 10 Gbps. It is connected to the host through 8 lanes of PCIe 2.0. To the network, it can be connected through 2 independent interfaces. Compared to its older sibling, 82598, it supports LRO and SR-IOV (among other improvements).

Its full specification [15] is publicly available for download from the vendor web portal. The specification covers both the supported features and their configuration interface for the controller driver. Unless otherwise specified, the information in this chapter is sourced from there.

In the Linux kernel, the driver responsible for this controller is called `ixgbe`.

3.1.1 Checksum Offload

The controller supports calculation of the IP and TCP/UDP checksums. However, the pseudo-header checksum must be computed by the software. The support is not generic, as the controller always inserts the checksum at offset 6 (UDP) or 16 (TCP) bytes from the beginning of the transport layer packet. The offset to the transport layer packet is configurable though, so the driver could exploit this to emulate the generic mode [15, § 7.2.5]. The Linux driver does not do so. SCTP CRC32 computation and validation is supported as well.

On the receive side, the controller supports IP and TCP/UDP checksum validation only for bare TCP or UDP over IP packets, no tunnels are supported [15, § 7.1.11].

3.1.2 Segmentation Offload

The controller supports TCP and UDP segmentation for transmission. However, the UDP segmentation is not implemented as IP fragmentation of a single datagram, but as splitting of the UDP datagram into more datagrams. The maximum size of a packet to be segmented is 256 KB. No advanced tunnelling can be involved, the engine can handle only up to two VLAN headers.

Receive Segment Coalescing (RSC), which is Intel's name for LRO, dynamically keeps at most 32 flow contexts per port, and coalesces TCP segments of those flows. RSC can be turned off and on individually for every receive queue. Interestingly, the controller is unable to coalesce TCP segments transmitted over IPv6 (while it supports their transmit segmentation).

3.1.3 Multiple Receive Queues

There can be as many as 128 receive queues configured. The majority of the receive pipeline is dedicated to queue selection [15, § 7.1.2]. First, hardware switching is performed, and a *queue pool* is selected. From there, the packed is examined by a variety of filters, which can select a concrete queue in the pool. In terms of configurability of the pipeline, the controller behaves differently when *Virtualization* is enabled. In this context, Virtualization refers to a mode where multiple software entities receive packets through the controller, and the controller offers additional features to support the use case. Otherwise, the available queues can be used to improve single-host performance.

Let us first explore the pipeline when Virtualization is disabled. No switching is performed, as all received packets are received by one operating system. The whole mechanism is devoted to selecting one of the 128 queues available. It is up to the software how it will assign the queues, yet few limitations exist, as we will see at the end of the pipeline.

The pipeline is constructed to consult a fixed sequence of tables. Every table can either hit the packet and select a queue index, or miss the packet. The first table that hits the packet determines the final queue number. At the end of the pipeline, there is a DCB[1] and RSS "filter", which hits every packet and determines the queue if it was not determined by the previous filters already.

RSS and DCB might not cover all the queues available. In that case, the driver can offload classification of packets and use the free queues to return the classification result.

Let us walk through the individual tables in the pipeline.

L2 Ethertype filter
Intended to steer packets of a specific ethertype to a particular queue. An example use case is an early classification of LLDP or IEEE 802.1X packets. This filter table is also used to mark the packet for other offloads (FCoE, IEEE 1588).

FCoE Redirection Table
Used to manually assign a queue based on 3 least significant bits of the

[1] Data Center Bridging

Fibre Channel Originator/Responder Exchange ID. As those IDs can be assigned uniformly, the table can serve as a FCoE-specific RSS.

L3/L4 5-tuple Filters
These rules match on any subset of the transport-layer protocol, the source and destination IPv4 addresses and the transport-layer port used. Unfortunately, the fields can match only concrete values and not e.g. network prefix of an address. The filter is useful to steer specific flows to a dedicated queue. There can be at most 128 such filters.

SYN Packet Filter
TCP packets with the `SYN` flag can be steered into a dedicated queue to mitigate `SYN` flood attacks.

Flow Director Filters
An advanced flow classifier. Apart from selecting the receive queue, the packet is marked with a tag configurable by software. Flows can be matched either exactly (max. 8 K filters) or by hashing the input values (max. 32 K filters). Matching is available for the VLAN tag, the IP version, the source and destination IP addresses, the transport-layer protocol, and the source and destination ports. Furthermore, the filter can match on any 2 bytes in the first 45 bytes of the packet (offset defined globally).

The matched flows can be dropped. The dropped flows can be either actually dropped or just redirected to a dedicated queue. The matched flows can also be tagged with a 15-bit software-selected unique identifier.

The amount of memory dedicated for Flow Director filters is configurable. The memory is shared with the receive buffer for packets.

When none of the previous filters matched and selected the receive queue, DCB and RSS takes place depending on the configuration. When the DCB mode is enabled, it extracts the 2 or 3 bits of the PCP field from the VLAN header (DCB assumes all packets are VLAN tagged) to select a *Traffic Class Index*. RSS computes the flow hash by a fixed algorithm (with configurable random key) and selects a configurable number of least-significant bits to compute an *RSS Index*. Those two indices are then used to compute a queue index. The software should refrain from using queues assignable by this algorithm for filter targets.

When Virtualization is enabled, the queues are distributed evenly among queue pools, with high-order bits of the queue index defining the target pool. Virtualization in this context does not necessarily mean only SR-IOV – the controller allows a different mode called Virtual Machine Device Queues (VMDq), which can be seen as switching performed only to select the high-order bits of the receive queue. This mode is intended to be used along with a software switch, which can use the classification information. In the SR-IOV mode, queue pools correspond to virtual functions.

The inner switch does not learn. Instead, it consults a multitude of tables to fill the target pool list. Among others, it consults the destination MAC address with respect to unicast, multicast and broadcast tables separately.

The switch operates in one of two modes – with replication enabled or disabled. Replication allows to copy the received packet to multiple queue pools at once. If

replication is disabled, the software is responsible for distributing packets among multiple targets, because the packet will be received by one queue only. In this mode, the inner switch is used purely for classification. When replication is enabled, the software can configure which pools will receive broadcasts, which multicasts and so on. This mode is better suited for use with SR-IOV.

3.1.4 Multiple Transmit Queues

For transmission, the controller also opens 128 queues. However, not all of them are scheduled in every configuration. Depending on the use of Virtualization and DCB, only 64 queues might be dequeued.

Either way, two scheduling phases are performed. First, the packets are dequeued from the 128 queues to at most 8 packet buffers. Then, the packets are taken out from the buffers and sent to the MAC for transmission.

Queues are distributed between queue pools (virtual functions) and traffic classes. When there are multiple queues dedicated to one traffic class inside a pool, they are always dequeued in a frame-by-frame round-robin order. Queues are distributed between classes to reflect class priority (high-priority classes have fewer queues than low-priority classes, because less traffic is expected to be buffered.)

The scheduling algorithms differ for every mode of operation, and their description here would be a mere copy of [15], Section 7.2.1, and therefore is omitted.

From the offloading point of view, in majority of modes the controller employs a weighted-strict-order scheduling of different traffic classes. This fact could be used to offload priority scheduler with an appropriate number of bands.

3.1.5 Other offloads

Apart from the classification offloads mentioned, the controller features two security offloads: LinkSec (MACsec) and IPSec. For both of them, the software must establish the Security Associations itself, and then install them to the hardware tables. The controller is capable of offloading AES-128. Both authentication and encryption is supported, provided they are using the same SA (IPSec only). [15, §§ 7.8, 7.12]

Another inline functionality that is emphasized in the specification is the FCoE support. The controller offers additional functions like Fibre Channel CRC computation or FCoE segmentation. [15, § 7.13]

3.2 Intel Ethernet Controller XL710

Although the XL710 is not a direct successor of 82599, but more like a member of another branch of Intel networking ASICs, they are similar in design principles and features.

The silicon was announced in 2014. It is embedded in adapters from Intel and Lenovo, plus adapters from less known vendors from China. As all other controllers from Intel, its specification [16] is publicly available.

The Linux kernel driver for this controller is called `i40e`, corresponding to the 40 Gbps speed. The patches introducing the driver were posted even before the specification was released, in 2013.

The controller offers two variants of connection to the network – either through 2 independent 40 Gbps ports or through 4 independent up to 10 Gbps ports. The ports can be connected either directly to the medium or to an external PHY using MAUI. The controller connects to the host through PCIe 3 with 8 lanes. A simple calculation shows that the maximum network bandwidth (80 Gbps) is higher than that of the PCIe interface (64 Gbps incl. overhead).

Part of the configuration presented here is stored in a non-volatile memory off-chip, and is therefore persistent most of the time. Such configuration changes the way how the device presents itself to the host and manages static allocation of resources. We can say that this configuration is part of the firmware, which is quite easily modifiable.

The controller presents itself as up to 8 physical functions. The important difference from virtual functions created with SR-IOV is that the drivers of the physical functions are in charge of configuring the virtual functions assigned to them. There can be as many as 128 virtual functions in total, arbitrarily divided among physical functions (configuration stored in NVM).

One could say that the multi-presence of physical functions introduces another layer of virtualization, when 8 almost-isolated environments can be created to run inner virtualized networks over physical wires shared by all of them. However, any of the drivers for the physical functions can request configuration of the global resources, such as the firmware and the non-volatile memory. So, the physical functions cannot be just handed out to customers to deploy their own virtual networks.

The controller supports a similar range of offloads as the 82599 controller. The big difference between them is the support for various tunnelling protocols across all the offloads, as the XL710 supports IP-in-IP, Teredo, IP-in-GRE, MAC-in-GRE (NVGRE), VXLAN and Geneve.

3.2.1 Stateless Offloads

Both checksum offloads and LSO are supported with the extended support for tunneling. Surprisingly, no form of LRO is supported.

Regarding the checksum offload for TCP and UDP, the pseudo-header must be computed in software. For MAC-in-UDP tunnels, the support does not cover the outer UDP header. Instead, those protocols are (un)supported in a generic way, because the software specifies only the total length between the outer and inner IP headers, allowing the inner transport-layer checksum to be computed without specifying the concrete tunnelling protocol in use [16, §§ 8.3.4.3, 8.4.4.3]. Similarly, the LSO engine supports TCP segmentation of packets up to 256 KB even when they are encapsulated in a tunnel.

3.2.2 Multiple Queues

The traffic to host is delivered through one of 1536 available queue pairs (a transmit and a receive queue form a pair). The distribution of queues is persistently

configured for physical functions, which can then dynamically assign them to virtual functions at runtime. While one physical function can be assigned all 1536 queues, virtual functions are limited to obtaining up to 16 queues.

The internal switch architecture does not work directly with functions but with entities called Virtual Station Interfaces (VSIs), which represent generic packet destinations or virtual switch ports. There can be as many as 384 VSIs, representing physical functions, virtual functions, switch control ports, traffic snoopers or just another target assigned to a physical function.

The internal switch configuration options are very rich. Just explaining the internal switch architecture occupies 150 pages in the specification [16, § 7.4]. We will try to avoid going into detail.

The basic idea is as follows. Whenever a packet is received on one of the physical ports, an outer VLAN tag (called service VLAN tag, or just S-tag) is stripped and determines an ID of the internal virtual switch that is then used for switching. This layer can be turned off, in which case the default internal virtual switch is used.

There can be as many as 16 internal virtual switches, each spanning a disjoint set of VSIs. Every switch can run either as a fully-featured manageable switch (allowing VM-to-VM traffic), or just port aggregator, which relies on external switch looping VM-to-VM traffic back. At most one physical port can be connected to every switch.

All the internal switches are not learning, and they must be managed in order to deliver any traffic. The forwarding database can be configured with rules matching on the Ethertype (for control traffic), the MAC address (optionally hashed), the VLAN ID, or the MAC address together with the VLAN ID. Individual ports can be marked as promiscuous for unicast and multicast traffic separately.

As the switching is realized on VSIs and not functions, the host can use the switching capabilities to offload a software switch without incorporating SR-IOV. Intel calls this mode of operation VMDq 2, which extends the VMDq mode we have seen in the 82599 controller. The key idea is to assign all concerned VSIs to the physical function and use the information created when the packet is switched to accelerate the switching in software. In this mode, the controller supports switching based on fields from up to the transport layer header – destination IP address, tunnel ID, inner MAC address in a tunnelled packet or a combination of these.

After the list of target VSIs is created, the packet goes through a series of filters very similar to those of 82599. We will not go through them again, let us just have a look at changes.

The engine was extended with support for all the tunnelling protocols. That especially means that the filters can be used on tunnelled packets (which miss all filters on 82599) and that the tunnelling header fields are available to match on.

The flexible field on which the Flow Director matches was extended to extract 16 bytes from up to 3 offsets within the payload of a packet [16, § 7.1.4]. The payload can be defined roughly as the first packet header that is not identified by the internal parser.

Parser identifies fields in the first 480 B of the packet. From these, a 128B field vector is constructed. The Flow Director filters are able to match on up to

48 bytes. At most 8192 rules can be inserted in total. The memory is shared for all functions – every function has a configurable portion of guaranteed space and the rest is available freely.

The RSS engine was extended to support multiple hash functions. Apart from the Microsoft-defined Toeplitz hash function (used in other RSS implementations), it supports a simpler variant when software needs to compute the value as well. Also, the maximum number of queues used for RSS was raised to 64, but for physical functions only, because virtual functions can have at most 16 queues each.

On the transmit path, the packet goes through a sequence of filters mainly provided for security purposes (anti-spoofing, validating VLAN tag, etc.). Then switching is performed to determine whether the traffic is local or needs to be scheduled to a wire. As the switching topology is guaranteed to be a tree rooted at the physical port, the controller can divide the available bandwidth hierarchically. Every switching element can be configured to divide its bandwidth between the entities connected to the element. At the lowest level, VSIs can configure how their bandwidth is distributed among their queues. Depending on the configured mode, the bandwidth is split among traffic classes – either at the root of the tree, under the switching elements or under VSIs.

3.3 Mellanox ConnectX-4

This controller from Mellanox was announced in late 2014. It is a combined network controller for both Ethernet and InfiniBand. The fourth version is not the most recent one, as Mellanox already produces ConnectX-5 and develops ConnectX-6. We decided to include ConnectX-4 mainly due to the Programmer's Reference Manual (PRM) [20] being public. However, the open driver for ConnectX-5 is already merged into the Linux kernel, and we will try to provide updated information where applicable.

Counterintuitively, the controller is driven by the `mlx5_core` driver in the Linux kernel, while the `mlx4` driver supports controllers up to ConnectX-3. The rationale seems to be that ConnectX-3 is a 40 Gbps controller, hence the suffix 4.

For complete configuration, an external collection of tools is needed. The kernel drivers are dedicated to controlling individual network functions, not the adapter as a whole.

The controller is capable of presenting up to 16 physical functions and up to 256 virtual functions using SR-IOV. Both of these need to be configured by the vendor-provided firmware utilities. Only then can the virtual functions be "activated" by standard means.

Connection to the host is realized using PCIe Gen 3 x16. To the network, the controller opens 4 or 8 SerDes[2] lines with 25 Gbps throughput each, which can be used to create an adapter with 2 physical 100 Gbps ports.

The global configuration of the controller is stored in non-volatile memory, and needs to be configured using an out-of-kernel utility. Such configuration includes mainly switching between the Ethernet and InfiniBand modes (if appli-

[2]Serializer-Deserializer. Such interfaces can be configured to support various PHY adapters.

cable), enabling SR-IOV and the number of functions created. Interestingly, the controller is capable of running one Ethernet and one InfiniBand port.

3.3.1 Switch Layout

While the PRM [20] provides enough information about the processing pipeline, it does not describe the operation of the packet processing before the packet reaches the PCI function. A brief explanation was provided in a cover letter for patches introducing the SR-IOV support for the kernel driver by Or Gerlitz [11].

The controller has two layers of switches. The first one, Multi Physical Function Switch (MPFS), is responsible for switching unicast traffic among physical functions. Broadcast and multicast traffic is always flooded. The MPFS is managed through so-called L2 Table, whose configuration interface is covered in the PRM [20]. Basically, the L2 Table matches packets based on the destination MAC address and optionally the VLAN tag, and selects the target physical function. Therefore, the MPFS is actually a very simple, non-learning switch.

The second layer, Ethernet Switch (E-Switch), exists for every physical function. It operates on entities called *Vports*. Initially, there are two Vports – one for the uplink, one for the physical function. Unmatched traffic always goes to one of these two, depending on the traffic direction. The E-Switch must be configured with rules to direct traffic to virtual functions as well. This is possible only from the driver of the physical function.

For configuration purposes, every driver is responsible for managing its own Vport context. Whenever the context is updated by a virtual function, the driver of physical function receives an event. It can check the validity and conformation to any local policy and then update the configuration accordingly.

3.3.2 Pipeline

The pipeline is made of simple elements which are chained together. For any of these elements, there can be multiple instances, resulting in significant flexibility. Instances are created at runtime through a command interface. Every instance can be configured separately.

A received packet arrives at a root *Flow table*. Flow tables are allocated in a TCAM and can match on all available fields. Those include fields from the Ethernet, VLAN, IPv4, IPv6, UDP, TCP, VXLAN and GRE headers. For MAC-in-UDP tunnels, inner fields are supported as well.

Flow tables contain *flow groups*. A flow group defines a mask for all the available fields, which selects bits that are required to match. A flow group then contains one or more *flows*, which define the matched value. A matched flow can either be passed to the next stage, forwarded to another flow table or dropped.

Newer versions of the controller extend the available actions to create a flexible match-action pipeline. The possibilities now include manipulation with tunnel headers (encapsulation, decapsulation), individual header modification or even IPSec encryption and decryption. We can see the new actions being used in the `parse_tc_nic_actions`[3] function in the `mlx5` driver.

[3] Linux kernel [30], file `drivers/net/ethernet/mellanox/mlx5/core/en_tc.c`, line 1859

To prevent cycles when forwarding is used, every table must have a level defined. Forwarding is possible only to a table with higher level. The root table is the only table with level 0.

Once in the packet lifetime, multiple target flow tables can be specified. From there, the packet is cloned and processed by multiple paths. The split is possible only when forwarding from a table with level < 64 to a table with level ≥ 64, enforcing the unicity condition.

To illustrate the simplicity of the engine in the background, the controller does not check whether the matched fields are valid in the context of the packet – for example, matching the destination IPv4 address does not automatically check whether the packet is IPv4. Unless the rules are programmed to check the protocol first, they might be matching on garbage.

Another action that can be performed only once in the pipeline is tagging the flow with an arbitrary *Flow tag*. The tag is then reported out-of-band in the completion event.

Once the last table is processed, the packet is forwarded to a specified Transport Interface Receive (TIR). This entity is responsible for performing stateless offloads, which will be discussed further. Also, TIR can perform RSS by selecting the target Receive Queue indirectly based on the output from a configurable hash function and a redirection table. With regard to available hash functions, the driver has similar flexibility as the Intel XL710 controller. Again, the driver is responsible for configuring the Flow tables so that only packets that are subject to RSS are delivered to TIR performing RSS. If the tables are misconfigured, the result of the hash function is undefined.

The receive queues are created dynamically as well. The responsibility of a Receive Queue is to store the packet and report to a Completion Queue, which may result in interrupting the host. Unexpectedly, the Receive Queue might strip a VLAN tag from the received packet first. We could not find this feature used in the Linux kernel driver.

For transmitting a packet, the pipeline is reversed. First, the packet reaches a Send Queue, where the packet is stored until it is to be scheduled. From there, packets are withdrawn by Transport Interface Send (TIS) instances, counterparts of TIR. TIS is responsible for performing LSO, if applicable. Also, a fixed priority is assigned to TIS, which presumably plays a major role in scheduling.

One would expect a match-action pipeline to be implemented on egress as well. It does exist, but at the egress of the E-Switch. Therefore, it is not covered in the PRM [20]. In the source code of the Linux driver, we can see the rules being constructed in `parse_tc_fdb_actions`[4].

3.3.3 Stateless Offloads

The controller supports checksum verification and calculation. As opposed to Intel controllers, it computes the pseudo-header checksum by itself and ignores the initial value of the checksum field.

Unexpectedly for such an advanced controller, all the stateless offloads are supported only for pure TCP/UDP over IPv4/IPv6, no tunnelling headers must be involved. This limitation is remedied in more recent versions thanks to the

[4] Linux kernel [30], file `drivers/net/ethernet/mellanox/mlx5/core/en_tc.c`, line 2386

possibility to peel off the tunnel headers before they reach the TIR, or add them after they are processed with TIS.

3.4 Chelsio Terminator 6

The controller ships in many variants, all of which enable a subset of functions. While that probably allows Chelsio to set more suitable pricing, it is quite confusing. For example, we cannot say for sure that any two features described here can be used at the same time.

The architecture of the ASIC is best described in a paper published by Chelsio [6], it however does not go into much detail. As the vendor refused to share any information with us, we have to resort to guessing. Many details are exposed in the manual for the proprietary driver extension and configuration utility [5]. Still, both manuals only explain the capabilities of the controller from the user's point of view, not the hardware capabilities in detail. For this kind of information, we had to carefully examine the source code of the Linux kernel driver.

The driver in Linux kernel, `cxgb4`, is shared for all controllers from the Terminator series, starting from Terminator 4. Therefore, we know the union of the capabilities of all these controllers, because we cannot differentiate them. As we do not expect features to be removed in newer versions, we suppose that the current state reflects the capabilities of Terminator 6.

As usual for modern high-end NIC, SR-IOV is supported. The controller handles up to 256 virtual instances, mapped to virtual or physical functions. Even though the controller can identify itself as 8 physical functions, only the first 4 are capable of SR-IOV. With the factory firmware, each of them can control 16 virtual functions. However, some controller variants (probably differing only in firmware) can be configured with up to 62 virtual functions per physical function, giving us the maximum of 256 functions in total. Unfortunately, neither the available documents nor the source code gives hints about the inner switch functionality.

3.4.1 Stateless Offloads

The controller supports all of the stateless offloads. An interesting requirement is placed on the software in case of IPv4 header checksum for LSO of MAC-in-UDP tunnels – the driver has to compute the checksum of the outer IPv4 header without the total length field, which is different for the last segment created.

For the receive checksum, the controller is capable of computing the checksum of the received packet in a generic way, as described in Section 4.2.1. It does so as the first controller from our list.

As for the tunnelling support, it seems that VXLAN and Geneve is supported for both checksum and segmentation offloads. Even though GRE is defined as one of the constants for supported tunnels, it is not used elsewhere in the driver code.

3.4.2 Match-Action Offload

There is a step in the pipeline where custom classification is performed and programmable actions are taken. The controller always contains a TCAM to install

up to 496 rules, and optionally also memory dedicated for half a million hashed rules. The action part might seem a bit constrained at first, but in the end it is quite powerful.

The match-action pipeline starts with a fixed parser, extracting a similar set of fields we have already seen – the MAC addresses, Ethertype, 2 layers of VLANs, IP addresses, TCP and UDP ports. In the kernel, the available fields are represented in the `ch_filter_tuple`[5] structure.

From these fields a compressed vector is constructed to be matched on. The compressed vector contains only the IP addresses and the L4 ports, extended with up to 36 bits of the fields from above. It is very important to note that the selection of bits creating the match vector is global and changeable only by reinitializing the controller.

The compressed vector is used to look up an entry in a table, be it a TCAM or a hash table. In both cases, the software representation of the entry is the `ch_filter_specification`[6] structure. From its layout, we can clearly see the possibilities.

An interesting difference from the other vendors is that there is a fixed number of slots for filters, and the slots are allocated by the software. The order of filters is not arbitrary, as filters with lower indices have higher priority.

For matched flows, exactly one of three actions can be taken: pass, drop or switch. When the packet is passed, its receive queue can be selected (otherwise it is selected by RSS automatically). More possibilities open with the switch action. First, the switch action instructs the controller to loopback the packet to a port. However, it can also modify some header fields. Namely, it can alter the MAC addresses, push, change, or pop the VLAN tag and/or modify the IP addresses and the L4 ports. That means the controller is capable of offloading e.g. routing with stateless NAT.

3.4.3 Other Offloads

Chelsio emphasizes a lot of other offloads the controller supports. The range of additional functions the controller features is:

- TCP Offload Engine (TOE),
- Remote Direct Memory Access (RDMA) over iWARP offload,
- Both target and initiator offload of iSCSI and NVMe-oF,
- FCoE initiator offload,
- Crypto offloads (IPSec, TLS, SMB, . . .),
- Open vSwitch offload,
- Bonding and active failover offload.

If the NIC driver had to support all of these, the driver would have to be an ugly hybrid driver spread across the whole kernel. It is not the case, instead, the upper-layer drivers (such as the iSCSI kernel implementation) communicate with the controller through the NIC driver using a network-like protocol, therefore the NIC driver itself does not care much about the extended features.

[5] Linux kernel [30], file `drivers/net/ethernet/chelsio/cxgb4/cxgb4.h`, line 1009
[6] Linux kernel [30], file `drivers/net/ethernet/chelsio/cxgb4/cxgb4.h`, line 1042

3.5 Netronome NFP-6000

As the last examined controller, we have chosen Netronome NFP-6000. The design of the NFP controller series is presumably completely different from the other vendors. Instead of being a highly specialized packet processing silicon, the controller is a programmable parallel compute unit with features for packet processing and network connectivity. The Netronome controllers NFP-4000 and NFP-6000 share the same architecture, but differ in the number of functional units used.

The information in this section is based mostly on papers published by Netronome [23, 24, 29], with a few bits deduced from the Linux kernel source code.

The controller is used exclusively in the adapters by Netronome. It is attached to the host via up to four independent PCIe Gen 3 x8 interfaces, connecting to up to four CPU sockets in one host. They can handle up to four 100 Gbps Ethernet interfaces, with both integrated MAC or a transmitter connected via SFP+, QSPF or CXP.

The controller is composed from so-called islands, which have isolated responsibilities. Most of the islands are connected with a bus. The architecture is modular, allowing to create different versions of the same controller design.

The islands include:

Ingress MAC and Packet Processing
: Receives packets from the network interface. Parses headers, verifies checksums and constructs packet metadata. The packet payload is sent to the memory units, the packet metadata to the Flow Processing islands.

Flow Processing
: Performs arbitrary processing of packets using the packet metadata.

Egress Packet Processing and MAC
: Reorders packets from the same flows and schedules them to the network.

ARM Subsystem
: Contains a fully-featured ARM processor, which is able to run Linux. This processor can be used to configure or monitor the controller as well as run any other application.

Crypto
: Specialized circuits to support the Flow Processing units in encryption and decryption of arbitrary data.

Memory units
: Globally-accessible memory units to store tables or data into. Apart from up to 30 MB of on-chip memory, there is an external memory unit which supports up to 24 GB off-chip DDR 3.

PCIe
: Used for communication with the host. Packets sent by the host are processed similarly to ingress packets, the payload is stored in the memory units and metadata is passed to Flow Processing islands.

An important building block is the Flow Processing Core (FPC), which is a programmable 32-bit processor core designed for packet processing. The core runs up to 8 threads, which are cooperatively scheduled when waiting for data, similarly to threads on GPUs.

The flow processing islands are made of 12 FPCs each. They do the most of the packet processing work. Apart from forming the Flow processing islands, FPCs are present in lower numbers in other islands as well, making even the fixed parts of the pipeline programmable.

The FPCs can be programmed using an open source SDK to perform any processing. The SDK provides a framework to program packet processing using the P4 or C language, or allows to write programs running on bare metal.

The processor is not fully featured, it has a simple architecture (e.g. cannot calculate with floating point numbers, there is no stack and so on), but certainly is more flexible than a match-action pipeline. The controller is therefore able to do any packet processing for any application, provided the program fits into the instruction memory.

If not configured and programmed by the user, the controller ships with firmware that emulates the behavior of a conventional NIC. The firmware offers several "apps", which define the capabilities of the controller from the operating system point of view.

As the capabilities of this controller are implemented by software, it does not make sense to describe the controller in terms of the current firmware version. We may assume that the controller fits well into any reasonable offload model.

Taken to the extreme, the controller is capable of running Open vSwitch by itself, on the ARM processor, offloading the heavy lifting to the FPCs, both without the host intervention. The ports of the virtual switch still correspond to individual network interfaces presented to the host, creating a very unusual scenario of a separate machine running Open vSwitch, connected via PCIe to the host.

4. Hardware Offloading in Linux

Before we dive into how Linux supports various offloading techniques and how they are configured and implemented, let us recall the development model of the networking stack in Linux kernel. Linux community does not publish any guidelines for network device vendors like e.g. Microsoft does with its Network Driver Interface Specification (NDIS) [21]. In the Windows world, vendors create controllers with the driver interface in mind, and are restricted to features specified by NDIS. In the Linux world, both the drivers and the driver interfaces follow the design of the hardware. The development is spontaneous and lacks centralized control. Therefore, all available solutions are somewhat improvised by definition.

4.1 Ethtool

Until recently, the only gateway to the hardware offloading features was the `ethtool` utility. Its main purpose is to communicate with and control the network device drivers. The name might suggest it is restricted to drivers of wired Ethernet adapters, but it is not. It can be used to control e.g. WiFi drivers as well. Before we review the implementation of various offloads, let us have a look at the tool usage and output.

The general syntax for `ethtool` is as follows:

```
1  $ ethtool [action] <ifname> [arguments]
```

Here `<ifname>` represents the (required) name of the network interface, and `[action]` selects a particular functionality of the tool, which can be further parameterized by `[arguments]` (both optional).

The utility is developed along with the kernel and shares the kernel versioning. It should be available in all Linux distributions.

4.2 Features

For every network interface, there is a set of flags called *features*. The flag set serves as a shared data structure at multiple places in the kernel. Most notably, the driver uses them to report the capabilities of the NIC, and the network stack to control them individually.

The `ethtool` command can be used to display the current state of all features with the `--show-features` action, as shown on Listing 4.1. In the first column, we can see all features that are supported by the current kernel. Next, the output shows which features are currently enabled on the interface by listing them as being `on`. In case the feature is supported but disabled at the moment, it is shown as being `off`. In case the feature is not supported, it is listed as `off [fixed]`. The last case, `on [fixed]`, is for features that cannot be disabled, because they do not correspond to offloads but rather to general-purpose properties, such as the ability to use high memory for DMA.

```
$ ethtool --show-features eth0
Features for eth0:
rx-checksumming: on
tx-checksumming: on
        tx-checksum-ipv4: off [fixed]
        tx-checksum-ip-generic: on
        tx-checksum-ipv6: off [fixed]
        tx-checksum-fcoe-crc: off [fixed]
        tx-checksum-sctp: off [fixed]
scatter-gather: on
        tx-scatter-gather: on
        tx-scatter-gather-fraglist: off [fixed]
tcp-segmentation-offload: on
        tx-tcp-segmentation: on
        tx-tcp-ecn-segmentation: off [fixed]
        tx-tcp-mangleid-segmentation: off
        tx-tcp6-segmentation: on
udp-fragmentation-offload: off
generic-segmentation-offload: on
generic-receive-offload: on
large-receive-offload: off [fixed]
rx-vlan-offload: on
tx-vlan-offload: on
ntuple-filters: off [fixed]
receive-hashing: on
highdma: on [fixed]
rx-vlan-filter: off [fixed]
vlan-challenged: off [fixed]
tx-lockless: off [fixed]
netns-local: off [fixed]
tx-gso-robust: off [fixed]
tx-fcoe-segmentation: off [fixed]
tx-gre-segmentation: off [fixed]
tx-gre-csum-segmentation: off [fixed]
tx-ipxip4-segmentation: off [fixed]
tx-ipxip6-segmentation: off [fixed]
tx-udp_tnl-segmentation: off [fixed]
tx-udp_tnl-csum-segmentation: off [fixed]
tx-gso-partial: off [fixed]
tx-sctp-segmentation: off [fixed]
tx-esp-segmentation: off [fixed]
fcoe-mtu: off [fixed]
tx-nocache-copy: off
loopback: off [fixed]
rx-fcs: off
rx-all: off
tx-vlan-stag-hw-insert: off [fixed]
rx-vlan-stag-hw-parse: off [fixed]
rx-vlan-stag-filter: off [fixed]
l2-fwd-offload: off [fixed]
hw-tc-offload: off [fixed]
esp-hw-offload: off [fixed]
esp-tx-csum-hw-offload: off [fixed]
rx-udp_tunnel-port-offload: off [fixed]
rx-gro-hw: off [fixed]
```

Listing 4.1: An `ethtool` command showing features of an Intel I219-LM NIC on kernel 4.16.

For features that can be turned on and off from the userspace, one can use the `ethtool` utility as well, namely with the `features` action. To give an example, the following command can be used to enable `ntuple-filters` and disable `generic-receive-offload`.

```
1 # ethtool --features eth0 ntuple on gro off
```

Unfortunately, the feature names differ from those listed by `--show-features`. To further confuse the user, not even the manual reveals the mapping between the names explicitly, only a textual description is used.

In the kernel source code, the feature flags are defined as macros in the `include/linux/netdev_features.h` file. Throughout the following text, we will reference the features using these macro names.

4.2.1 Checksum Offload

Linux makes use of checksum offloading. As there are no knobs to tune the behavior, the checksum computation is either offloaded or not. The only means of configurating the checksum offload are the feature flags. However, it would be hard to cover all the possible cases which the hardware can offload with any static description – imagine all the combinations of tunnel headers, VLAN tags, IPv6 extension headers, etc. Instead, the stack handles offloading the checksum computation on a per-packet basis.

An extensive documentation of checksum offloading in Linux can be found in `include/linux/skbuff.h`. A short overview follows.

The `sk_buff` structure carries multiple fields related to checksumming. Most notably, the `ip_summed` field indicates the current state of the `sk_buff` with respect to checksum. The meaning of its values differs for packets being sent and received.

Ingress Direction

First, let us explore the receive path, which is simpler. The dedicated feature flag `NETIF_F_RXCSUM` is used to control receive checksum offload in the driver. However, the stack does not rely on driver behavior, it always examines the `ip_summed` field. When the driver receives a packet from the NIC, it is too late to change the state of an offload. Therefore, the driver is expected to use the meta-information given by the device to detect what checksums were verified, and modify the `ip_summed` field accordingly. The options are (not in original order):

`CHECKSUM_NONE` the device did not perform any kind of validation.

`CHECKSUM_PARTIAL` the packet comes from a virtual source with offloaded transmit checksum, therefore the checksum is not expected to be verified.

`CHECKSUM_UNNECESSARY` some of the checksums were verified to be correct by the device. The zero-based index of the last verified checksum is denoted in the field `csum_level`.

CHECKSUM_COMPLETE the device computed the whole packet checksum, which the driver fills into the `csum` field.

From the point of view of the stack, the most flexible and future-proof option is `CHECKSUM_COMPLETE`. As the stack has to parse the headers anyway, it is easy to compute their checksums and subtract them from the computed checksum. Therefore, the NIC accelerates the verification in all layers without having to understand them.

Egress Direction

The transmit path is a bit complicated with the presence of Generic Segmentation Offload (GSO), which we describe in the next section. For now, let us focus on plain checksum offloading without GSO. In contrast to the receive path, when the packet leaves any software entity (networking stack, driver), the entity must ensure that the packet already has a valid checksum or the following entity is prepared to compute it. The driver indicates its ability to compute checksums by the feature flags. There are five of them:

NETIF_F_IP_CSUM and NETIF_F_IPV6_CSUM indicates the ability to compute one's complement checksum for TCP/UDP over IPv4 and IPv6, respectively. Deprecated in favor of `NETIF_F_HW_CSUM`.

NETIF_F_FCOE_CRC and NETIF_F_SCTP_CRC indicates the ability to calculate CRC for FCoE and SCTP packets, respectively.

NETIF_F_HW_CSUM indicates that the driver can compute any one's complement checksum as defined by the fields in the `sk_buff` structure.

As for the configuration, the `ethtool` command controls all the flags at once, it is not possible to selectively enable or disable only a subset of the available checksum offloads using the current userspace API.

We can see that none of the flags consider the checksum of the IPv4 header. The reason for that is simple – it is not expensive to calculate the checksum for the 20 bytes of the header, especially when the header is constructed in the software already. An obvious exception is an IP packet emitted by TSO, where the IP packets are constructed by the controller itself.

We can also see that there were attempts to cover the simplest cases, which were then superseded by the generic one's complement checksum capability. The idea is that the driver can check whether the controller will be able to compute the checksum. If the combination of headers is recognized by the controller, the driver instructs it to do so. In the other case, it just computes the checksum in software. Still, the driver must be prepared to accept packets which are already checksummed (or does not require any checksum to be computed in general).

The slice of a packet to be checksummed in the generic cases (including FCoE and SCTP) is defined as a suffix of the packet starting at the position specified in the `csum_start` field of the `sk_buff` structure. The driver shall ensure that the checksum will be written at offset `csum_offset`. In case the controller does not support computing the checksum in a generic way, the driver should check the values in these fields to make sure they will be recognized by the controller.

In order to simplify the driver code, a helper `skb_csum_hwoffload_help` is provided. Whenever the driver cannot be sure that the controller will compute the checksum, it can just call the helper and it will compute the checksum in software.

Seemingly, the situation complicates when tunnelling is incorporated, as there are two headers which include checksums – e.g. an inner TCP packet wrapped in an outer UDP packet (with IP and MAC headers in between). While it might happen that the TCP checksum is already computed because the tunnel is only encapsulating traffic from elsewhere, it is definitely possible that the traffic is local and therefore both checksums need to be filled in.

There is a surprising clever trick. When the checksum of the outer packet is computed, it is defined as a sum of partial checksums of all its parts. The part that is most expensive to compute is the inner TCP packet with the payload. However, since the inner packet carries its own checksum, the checksum of the whole inner packet including the checksum field is not affected by its contents. This means that the checksum field for the outer packet does not depend on the inner TCP packet at all. Thus, the outer checksum can be computed inexpensively in the software from the headers only, and the hardware checksum offload can be used to compute the expensive inner checksum. This technique is called Local Checksum Offload and was implemented by Edward Cree [7].

4.2.2 Segmentation Offload

The Linux networking stack utilizes various segmentation offloads. Moreover, it pushes the idea even further, and implements software techniques to reduce the number of stack traversals. In the end, these software techniques naturally extend to the hardware-offloaded techniques.

Quite isolated is a utilization of LRO. Again, once the driver receives a packet, LRO is already done. Therefore, the only support which is needed from Linux is a way to configure whether LRO should be enabled or not. As expected, the `NETIF_F_LRO` feature flag serves exactly this purpose.

When LRO is not available or has to be disabled (e.g. because of routing), there is still a way to coalesce packets to move data around in bigger chunks. Linux implements so called Generic Receive Offload (GRO). Before we explain it, it is necessary to introduce the NAPI.

NAPI is a mechanism of the Linux kernel that reduces overhead induced by interrupts. When a network device receives a packet, it copies it to a prepared DMA buffer in the host memory, marks that buffer as valid, and interrupts the host. A NAPI-compatible driver then does not receive the packet in the interrupt handler itself, but schedules a polling softirq handler instead. Most importantly, it disables the interrupt temporarily. The initial packet as well as those received in the meantime are processed in a polling loop. When no more packets are available, the polling loop ends and enables the interrupt again. This way, packet processing is not being interrupted by reception of new packets, resulting in considerably higher packet processing rate.

When the packets are already received in batches, the GRO mechanism works to merge them if possible. As the `sk_buff` structures are received by NAPI, a `gro_list` of `sk_buff`s is built. Any newly received packet is first compared

with the members of the list, checking whether the two are similar enough to be merged into one.

An important feature of GRO is that it is not limited to any particular protocol layering. The mechanism is fully generic (hence the name), and individual protocol handlers might decide what information can be lost by merging packets. As a rule of thumb, packets that are candidates for merging must contain the same sequence of headers and only a few selected fields might differ.

The received packets are kept in the `gro_list`, until they are delivered (passed to the upper layer), which happens once any of the following condition holds:

- The GRO protocol handler decides to deliver a packet. This happens for example when a TCP packet with any flag arrives, because flagged packets cannot be coalesced. Then both packets are delivered immediately, in the correct order.
- The `gro_list` would become too large (more than 8 entries in the current kernel). Then the oldest packet is delivered to make room for the new one.
- The NAPI polling loop is over. Then all packets are delivered at once.

In contrast to LRO, GRO does not coalesce packets in a lossy way. Merged packets not only have to belong to the same flow, but also have similar characteristics like their timestamp. The goal is to allow GRO-merged packets to be later split into the same segments on output. Therefore, GRO can be used on routers and bridges as well.

Some NIC vendors, being aware of LRO limited usability, implemented a more strict version of LRO, which passes enough metadata about the original segmentation of packets to re-segment them later. These NICs can then offload GRO to the hardware. Examples include recent Broadcom and Qlogic controllers. This feature is indicated by the `NETIF_F_GRO_HW` feature flag.

The memory layout of the `sk_buff` structures allows for creation of fragmented buffers. This feature is used heavily in GRO, as the packets are not copied into one big continuous buffer – instead, their fragment lists are concatenated, which is a constant operation.

The counterpart of GRO is Generic Segmentation Offload (GSO). The GSO mechanism segments a super-packet just before it is passed to the driver – so that the driver code is not complicated by segmentation. The super-packets might come from GRO, or be directly created from the data sent through a socket.

As we have seen, there are NICs that can segment the packet in the hardware, provided its protocol layering is compatible. When this is the case, the network device indicates the situation using a feature flag, and the segmentation is not performed. We can see TSO as one of the special cases here.

There is yet another technique that sometimes allows to offload segmentation of tunnelled packets to hardware without specialized hardware support. The basic requirement is that the hardware must support segmentation, where the outer headers are just bytewise copied.

Suppose that all segments of the packet (including the last one) are the same size. Then, none of the outer header fields need to be changed when the inner packet is segmented, including the outer checksum. The idea is similar to Local Checksum Offload, checksum cancels out the changes in the inner packet, making the outer checksum constant.

The initial requirement of having equally-sized packets is easy to achieve. Instead of giving up on late segmentation, the payload is split in two. The first is sized to an integral multiple of MSS, the second holds the last segment of the different size.

Unfortunately, the IPv4 ID will be the same for all segments. However, TCP streams require IPv4 packets to carry the "Don't Fragment" flag, therefore the ID should not be examined by any network device, as proposed in RFC 6864 [31].

Because the segmentation is only partially offloaded, this feature is called Partial GSO. It is controlled by the `NETIF_F_GSO_PARTIAL` feature flag.

4.2.3 TCP Offload Engine

In Linux, full TCP stack offload is not supported at all. It is not just unsupported, it is actively rejected. There are many reasons for that, summarized in the article about TOE at Linux Foundation Wiki [17]. The most relevant reasons for doing so include low flexibility in supporting the solution, complicated updates and possible security flaws.

4.3 Multiple Queues

Regarding the reception of packets into multiple queues itself, there is not much the network stack can do. The packet itself is received by the NIC driver, which constructs an `sk_buff` structure and hands it out to the stack for further processing. Similarly, using multiple hardware queues for transmission is nothing complicated. However, the idea to scale the number of channels that process the packets lead to several techniques.

As a quick way to see how many queues are available and being used, one can use the `ethtool` command:

```
$ ethtool --show-channels eth0
Channel parameters for eth0:
Pre-set maximums:
RX:                 0
TX:                 0
Other:                 1
Combined:         63
Current hardware settings:
RX:                 0
TX:                 0
Other:                 1
Combined:         63
```

The numbers represent the maximum and currently configured number of queues. If the device allows to use a different number of receive and transmit queues, they are given in the `RX` and `TX` rows. When the device requires to instantiate queues in pairs, one for each direction, the numbers are listed as `Combined`.

4.3.1 Receive-Side Scaling

First, let us consider pure RSS. To benefit, the card must be able to allocate and use as many hardware interrupt vectors as there are queues. Also, the interrupt vectors must be pinned to individual CPU cores. This is done automatically by the NIC driver, as the interrupt vectors have to be registered even if RSS is not enabled.

In a sense, RSS goes directly against the effort of NAPI to reduce the number of interrupts. Instead of packets being handled in batches, they are spread out to multiple queues, which trigger individual interrupts to be processed. To maximize the overall throughput, is most beneficial to configure one receive queue per CPU core.

RSS is configured through the `ethtool` utility, namely its `--rxfh` action. The current setup can be queried with the `--show-rxfh` action.

4.3.2 Receive Packet Steering

Similar to software segmentation offloads, the Linux network stack utilizes a software variant of RSS called Receive Packet Steering (RPS). The key idea of both mechanisms is to distribute packet processing to multiple cores as soon as possible. When RPS is enabled for a given receive queue, then every packet received by that queue is hashed and redirected to a CPU determined by the hash. This happens as one of the first things after the packet exits the NIC driver, in `netif_rx_internal`[1].

RPS cannot be as beneficial as RSS, as the CPU handling the packet first has to parse the packet headers just to compute the hash. On the other hand, it is by far more flexible than the hardware implementation, as it can handle multiple protocols, tunnelled packets, various hashing functions and so on. It is also hardware independent, and can work even with devices with a single hardware receive queue.

Unless handling mixed traffic that is only partially supported by RSS or when the number of queues supported by RSS is significantly lower than the number of CPU cores, it makes little sense to enable both RSS and RPS.

RPS in Linux is configured for every receive queue separately through `sysfs`. In the directory of the queue, e.g. `/sys/class/net/eth0/queues/rx-0/`, there is a file named `rps_cpus`, which contains the hexadecimal representation of a bitmap of CPU threads where to redirect the packets using RPS. It is disabled by default, one can enable it by setting bits in the bitmap.

4.3.3 Receive Flow Steering

As a third offload technique with similar acronym, Linux implements Receive Flow Steering (RFS). The idea behind it is very simple. Instead of moving the packet processing (and the application) to the CPU where the packet was received, redirect the packet to where the application is running and the packets are processed. It is important to understand that while it might be worth moving

[1] Linux kernel [30], file `net/core/dev.c`, line 4018

the application once, the scheduler might need to migrate it elsewhere to balance load.

When configured, RFS essentially only extends the lookup mechanism after the RPS hash is computed. The lower bits of the hash are used to find an entry in the global `rps_sock_flow_table`. If the entry belongs to the flow being examined, RFS steers the packet to the CPU given in the entry instead of falling back to RPS.

Entries are added to the flow table by protocol layers, with every packet processed or awaited by calling `sock_rps_record_flow`. Entries are never explicitly removed, instead, they are being replaced by new flows with the same low-order bits of their hash. The size of the table is configurable through the `net.core.rps_sock_flow_entries` sysctl variable.

As there might be multiple CPUs waiting for the packet, the target CPU field might change quickly, potentially delivering newer packets earlier than older ones. To overcome this issue, two layers of tables are involved. The layer that is actually used for steering is local to the receive queue, and the target CPU for a flow in that layer only changes when no packets of that flow are waiting in the current CPU queue. The size of the local table is configurable through a the `rps_flow_cnt sysfs` file in the queue directory. A more detailed description can be found in the message of the commit implementing RFS [13].

As we have seen, the controllers usually feature some classification mechanism which allows to select the receive queue based on packet headers, and therefore could be used to offload RFS to hardware. When the `NETIF_F_NTUPLE` network device feature is enabled, the `ndo_rx_flow_steer` callback is invoked with the information about the flow and the target receive queue whenever a new flow is to be steered. As receiving the packet to a wrong queue is not a serious error, there is no mechanism to remove a flow from the hardware. The hardware is expected to recycle older flow entries just as the software does.

Among the controllers we examined, only Mellanox ConnectX offloads RFS in Linux. Instead of replacing the flows based on their hash value, it uses entries in the Flow Table (as described in section 3.3.2) in a circular order.

4.3.4 Ethtool Network Flow Classification

While RFS improves cache locality, it does so only for flows that have been already seen. For example, if a dual-core system was running two virtual machines on dedicated CPU cores, the system would have to redirect half of the flows on average, because they would be initially sent to the wrong CPU core by RSS or RPS.

Some NICs (from our selection both the Intel controllers and Mellanox ConnectX) implement an optional `ethtool` operation to support explicit flow classification. Through the `ethtool` utility, these controllers can be configured to steer the matched packets to a specified receive queue. This mechanism has no software counterpart, therefore it is not considered to be an offload. If the mechanism is not implemented or enabled in the NIC driver, it just cannot be used.

The setup starts with the user invoking `ethtool` with the `--config-nfc` action. Several common header fields can be specified for the classification, among others the L4 port, the IP addresses, and the VLAN tags. Every rule then carries

an index of the target receive queue where the matched packets will be sent. If the queue is specified as -1, the packet is to be dropped by the controller.

Later, this mechanism was extended to store the target virtual function in the high-order bits of the queue number. This extension allows to select a receive queue of a different network device than the rules are installed to – quite drastically changing the purpose of the mechanism. However, this was not adopted by any other controllers than the Intel ones, because as we have seen, other controllers perform switching separately from selecting the receive queue.

4.3.5 Transmit Packet Steering

In contrast to various scaling techniques on the receive side, there is no need to scale on the transmit side. Packets are generated by applications, which are naturally run by available CPU threads (as defined by local policy). The only optimization the stack offers is Transmit Packet Steering (XPS).

The technique is essentially a careful hardware transmit queue selection based on the originating CPU. For every transmit queue, a set of CPUs which may use this queue for transmission can be specified. A reverse mapping is constructed and whenever a CPU needs to send a packet, a queue is selected using the flow hash, similarly to RPS. The queue number is then recorded for the socket so that next packets are sent using the same queue, preventing reorders.

XPS is mainly an optimization to moderate the congestions on queue locks.

4.4 Express Data Path

When talking about high-performance packet processing in Linux, we must mention the eXpress Data Path (XDP). The idea of XDP is to allow the user to inject an arbitrary program to process packets early in the stack, even before the `sk_buff` structure is allocated. The expected use cases include early packet dropping for DoS protection or wire-speed load-balancing or forwarding.

Obviously, allowing a user-specified machine code to run in the kernel context is usually considered a security flaw. In the case of XDP, a special restricted instruction set is used – the Berkeley Packet Filter. BPF is designed to be safe when executed in the kernel context. Most notably, BPF programs cannot contain backward jumps and therefore they always end after a finite number of instructions. Also, the instruction set is simplified enough so that programs can be verified whether they access only valid memory.

The initial usecase of BPF was to allow packet sniffers to filter packets before they are passed to userspace [18]. The user attaches a BPF program to a socket, the kernel verifies the program and runs it for every received packet. Depending on the result code, the packet is copied to the sniffing socket.

In Linux, the original BPF instruction set was enhanced to better suit modern processor architectures. Also, support for data structures like arrays, hash tables, and tries was added. Actually, the instruction set no longer resembles the original BPF, but the name has not changed. Sometimes, the enhanced set is called eBPF while the original cBPF (classic BPF). In the Linux context, BPF almost exclusively refers to eBPF.

There is an in-kernel JIT compiler from the enhanced BPF to the native instruction set. Therefore, running BPF programs does not come with any performance penalty. The compiler handles cBPF programs as well, by transcribing them to eBPF first.

The XDP programs are invoked as soon as a packet is received by the driver, before the `sk_buff` structure is allocated. That means that XDP needs to be explicitly supported by the driver, even though it does not depend on the hardware at all. The program executed in XDP can decide what to do with the packet using return codes. It can just pass the packet, in which case it is processed as usual. The packet can be also dropped, in which case the slot in the receive queue can be instantly reused with the same backing pages, reducing the overhead of dropped packets to a bare minimum. Finally, the packet can be modified and sent out using the same network device.

Unexpectedly, XDP can be offloaded to the hardware. Currently only the Netronome NFP controllers are able to do so, due to their software nature, but we can expect more controllers to support running BPF programs in the future.

4.5 Traffic Control

The Traffic Control (TC) subsystem exists for handling packets of different classes in Linux. Before we get to the most advanced packet modification offload techniques, we need to look briefly at how TC works.

One could say that the main purpose of TC is to assign a label (called *traffic class*) to every packet and then act on the packet depending on its traffic class. For example, treat packets with interactive traffic with priority.

Some manuals use the TC abbreviation for Traffic Class. To avoid confusion, we will always refer to Traffic Control with TC in the thesis.

The TC subsystem was designed to support the Differentiated Services architecture in its full flexibility – it allows to differentiate packets into classes, and apply policing, shaping, and scheduling. The subsystem is very flexible and generic, thus we do not aim to describe it completely.

The subsystem is configured via netlink. However, there are only a few userspace applications that control the TC subsystem. Usually, the configuration is manual, in which case the `tc` utility from the `iproute` package is used.

4.5.1 Queue Disciplines

The TC runtime configuration consists of a tree of Queue Disciplines, for every network device and direction of travel. In both the `tc` utility and all the available literature, Queue Disciplines are called qdiscs for short. We will stick to this convention.

Let us consider the egress direction first. There, the TC subsystem serves as the main buffer between the applications and the network interface. Whenever a packet is to be sent by an application, it is *enqueued*. When the driver is ready to emit a packet (there is an empty slot in the hardware queue), a packet is *dequeued* from TC. The implementation of these two operations is what defines a qdisc.

Take for example the `pfifo` qdisc, which is probably the simplest one possible. It is a simple FIFO queue, for which the enqueue and dequeue operations have the standard meaning. The number of packets in a queue is bounded and when the queue is full, new packets are dropped.

There are two flavors of qdiscs – *classless* and *classful*. The classful qdiscs constitute the inner nodes in the qdisc tree, as they have a child qdisc for every class of packets. The classless qdiscs are the leaves of the tree and actually store each packet until it is dequeued.

One would expect the classless qdiscs to treat all packets the same, but that is not true. Classless and classful in the TC context refer to the presence of child qdiscs, not differentiating classes of packets.

A typical example is the `pfifo_fast` qdisc, which is classless. It uses the TOS of the packet to select one of three bands[2]. Every band is a simple `pfifo`-like queue. The bands are dequeued in a strict priority order. Therefore, this classless qdisc prioritizes packets depending on their TOS field.

Another good example is the `tbf` qdisc, which is classful even though it has only a single child class. The purpose of the `tbf` qdisc is to shape the traffic dequeued from the inner qdisc using the Token Bucket Filter algorithm.

An alternative qdisc taxonomy is suggested by the subsystem author, Alexey Kuznetsov, in the short review in `net/shed/sch_api.c`, where he calls classless qdiscs "queues" and classful qdiscs "schedulers". The naming is not perfect either, because some "queues" (like `pfifo_fast`) can perform scheduling as well, while some "schedulers" (like `tbf`) might not. As most of the world seems to stick to the class-based taxonomy, we will use it as well.

Classless qdiscs can do more than just queue packets. For example there is the `red` qdisc, which implements the Random Early Detection algorithm to prevent congestion. In short, this qdisc randomly drops packets when it is filling up, with the expectation that the flows that might cause the congestion have higher probability to be selected, because they have a higher number of packets going through. Because a dropped packet is usually a signal for the sender to slow down (e.g. in TCP), this can prevent the flow from congesting the link early.

Classful qdiscs are what adds flexibility to the system. Their purpose is not to directly enqueue and dequeue packets, but to select a child qdisc to delegate the operation to. Classful qdiscs mostly differ in the dequeue operation implementation, as they define in what order the child qdiscs are dequeued, thus performing scheduling. For example, the `prio` qdisc dequeues inner classes in a strict priority order.

As another example, the `htb` qdisc can be used to divide the available bandwidth among multiple classes using the Token Bucket Filter algorithm, hierarchically (hence its name, Hierarchical Token Bucket). The exact scheduling algorithm is quite complex, but allows the classes to share bandwidth with guaranteed rates while allowing to steal the unused bandwidth from others.

The implementation of the enqueue operation might be interesting as well, but is usually dominated by the need to select the child qdisc based on arbitrary packet characteristics. For this purpose, the TC subsystem contains filters.

[2] Band is just another name for a traffic class, but let us avoid the term for a classless qdisc.

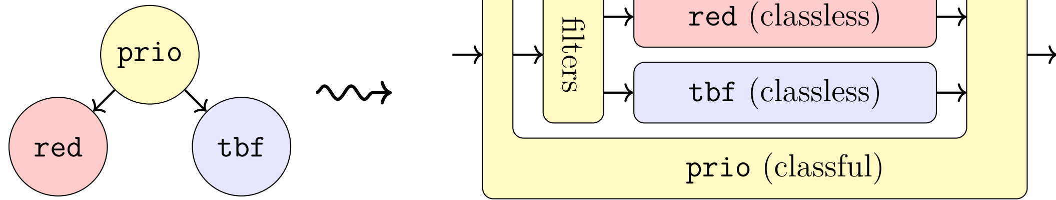

Figure 4.1: The composition of qdiscs. Filter block attached to the `prio` qdisc is used to select the inner qdisc. Inspired by figures in [1].

4.5.2 Filters

Filters are runtime instances of classifiers. Filters are attached to qdiscs and do the actual classification of packets. The separation of responsibilities between qdiscs and filters follows the Unix philosophy, where simple tools can be stitched together to create complex policies. Many classifiers are available to the user. If there is no classifier that would serve the purpose, the user is encouraged to implement a new one.

Filters are invoked to look at the packet and decide whether the packet belongs to some traffic class. Usually, filters directly select the child qdisc for the packet to be enqueued to. Multiple filters can be attached to a qdisc, in which case the priority assigned to them matters. Filters are always created for a single network protocol only – in particular, filters for IPv4 and IPv6 are not shared and must be instantiated twice if needed.

As one of the oldest classifiers, there is `u32`, the Ugly (or Universal) 32-bit classifier. It is able to match on any 32-bit word on the network layer. The match can also contain a mask, which selects the matched bits individually. Moreover, the classifier can use the masked word to index a hashtable. Implementation-wise, matching a concrete key and performing a lookup in a hashtable always alternates, but both can be trivialized to match everything in the step. The `u32` classifier tree can be created such that it is very efficient compared to combining other classifiers.

We would like to mention the `flower` classifier as well. This classifier uses the kernel flow dissector to extract the header fields and allows to match on them. Initially it was called `openflow`, because it was supposed to match on fields defined by the OpenFlow protocol [25]. As there is no reason to limit the classifier to these fields, it is extensible with other common header fields.

An important distinction between the `u32` and `flower` classifiers is that while `flower` uses header fields identified by the kernel flow dissector, `u32` requires the user to identify the headers and fields themselves. On the other hand, it is possible to use `u32` for proprietary protocols without modifying the kernel code.

4.5.3 Actions

Because the TC subsystem has the ability to classify packets, it seemed convenient to use the classification for more than just selecting the target traffic class. Ini-

tially, filters could return an action code, which could for example drop the packet immediately (TC_ACT_SHOT) or restart the classification (TC_ACT_RECLASSIFY). [3]

With the knowledge of filters and simple actions, we can finally introduce the ingress qdisc. As it makes little sense to shape or schedule incoming traffic, the TC subsystem has limited purview on the ingress side. Because there was no leaf qdisc which would actually not queue packets, a special no-op qdisc called ingress was created. The ingress qdisc cannot have children, but performs classification. Its only initial purpose was to classify traffic and perform policing – e.g. drop traffic exceeding the configured bandwidth.

Finally, a need for executing multiple actions arose. To satisfy it, another type of runtime entity was introduced – an action. In the source code, they are called filter extensions, but the user manipulates them as actions. After an action is executed, it returns the action code, as the filters initially did.

Some classifiers were extended to accept attaching multiple actions to them. When such a filter matches, the first action is executed. Another action result code was introduced (TC_ACT_PIPE), which cause the next action to be executed. This way, the user can program the subsystem to do a lot of packet processing in the kernel.

For example, there is the mirred action, which can perform packet mirroring or redirection. It can be used both to make the packet appear on ingress of a different network device, or to be sent out with one.

As a special action there is gact, the generic action. It does nothing except returning a specified action code, emulating the original functionality of filters.

Several other actions are available to modify the packet data or metadata, such as pedit, skbmod, skbedit, or csum. As a special case of packet modification, VLAN tags and tunnel headers can be added or stripped by the vlan, ife, or tunnel_key actions. The list is not exhaustive, new actions can be implemented.

The whole pipeline is not necessarily linear. There are action codes that make the execution engine jump between actions or restart the classification.

4.5.4 Actions on Egress

The classification-action pipeline allows the user to perform a lot of packet processing in the kernel. However, it required a classful qdisc in order to be able to execute filters and actions. Also, it was hard to configure TC for both QoS and packet processing at the same time.

Therefore, Daniel Borkmann introduced the clsact qdisc[3]. Essentially, it is an ingress qdisc to which two filter vlocks can be attached. One of them is used as the ingress chain as with the ingress qdisc, the other one gets executed for egress traffic with a new hook. This way, the TC subsystem is invoked twice for an egress packet – first for the egress chain in the clsact qdisc, second for the root egress qdisc.

[3] Actually, these were at first implemented by the policing framework of TC, and the macros were prefixed with TC_POLICY_. Policing was later generalized to actions.

4.5.5 Modularity

The TC subsystem is modular. Qdiscs, classifiers and actions can be distributed as kernel modules and loaded into the subsystem at runtime. They can even be developed separately from the mainline. Therefore, it is not possible to describe every possible behavior of the subsystem.

4.5.6 Offloading

As we have seen, there is a considerable overlap in the packet processing capabilities between controllers and TC. This resulted in what kernel engineers call TC offloading.

All the techniques described below are controlled by a single feature flag, `NETIF_F_HW_TC`. Unlike other features, this one is rather used to disable the behavior – enabling it might not change anything.

At first, TC offload was limited to offloading transmit priority scheduling to support QoS [10]. The network device gained a new callback, `ndo_setup_tc`, which was called to setup a number of traffic classes. The driver then configured the hardware scheduler and the stack with a mapping of classes to transmit queues. The stack then used the priority queues for priority traffic, improving latency by skipping the hardware buffers. The behavior was triggered by attaching an `mqprio` qdisc, which was created specifically for this purpose.

Quite recently, the mechanism was extended to also support other qdiscs, and most notably, classifiers and actions. So far, `mqprio`, `cbs`, `red` and `prio` qdiscs and `u32`, `flower`, `matchall`, and `bpf` classifiers are at least partially supported by some drivers.

The mechanism works as follows: whenever the TC configuration is modified in a way interesting for some driver, an event is generated. This event is announced to the driver through the `ndo_setup_tc` callback. The callback is now effectively just a joined callback for different events, which are distinguished by its argument of the `tc_setup_type` type.

In reaction, the driver parses the event data and decides whether it is feasible to offload what triggered the event. For example, the `ixgbe` driver offloads the `u32` classifier using the Flow Director filters described in section 3.1.3. The scheme of the offload is as follows:

1. The driver handles the block creation event (more about blocks later), in which it registers another callback.
2. The callback is invoked to handle `u32`-specific events, such as creation or deletion of inner nodes of the `u32` tree. The most interesting case is the creation of a key node (`tc_u_knode`), which performs matching of a concrete value, that is handled in `ixgbe_configure_clsu32`[4].
3. Even though the Flow Director rules support a flexible field, the offload does not consider it. Instead, it tries to match the hardware parser to the `u32` tree, identifying the matched fields. When the fields cannot be identified, the rule cannot be offloaded.
4. Actions for the rule are parsed in `parse_tc_actions` in the same file. We can see that the driver recognizes actions to drop the packet (as redirection

[4] Linux kernel [30], file `drivers/net/ethernet/intel/ixgbe/ixgbe_main.c`, line 8869

to the drop queue) and to redirect the packet to another function of the same device. If the action is not supported, the rule cannot be offloaded.
5. A Flow Director rule equivalent to the TC rule is inserted into the hardware table.

This generic procedure is followed by the drivers of all the examined controllers as well. Offloading the `flower` filters is a bit simpler – the driver does not need to parse the fields, because the classifier matches on well-known fields. The drivers vary mostly in the actions that are supported.

The mechanism works quite well as an offload. The user can use any qdiscs, classifiers and actions and provided the code is correct, compatible rules are offloaded. But what happens when only a subset of rules can be offloaded? As the offloaded rules are performed before they reach TC, how can the driver be sure that executing the rules in different order preserves the policy defined by the user?

The answer is unfortunate – it cannot. Currently, offloading a subset of rules can result in behavior which is different from that of TC in software. This is one of the biggest design flaws of the solution, which we try to address with the subsystem proposed in Chapter 5.

A partial workaround is provided by the ability to control whether individual rules can and will be offloaded. For relevant classifiers, two flags can be specified: `skip_hw` and `skip_sw`. When `skip_hw` is set, the rule will not be offloaded. When `skip_sw` is set, failure to offload the rule will result in the rule being rejected by the software as well. It is currently recommended to use these flags explicitly when TC offloading is enabled.

4.5.7 Shared Blocks

One of the root design features of TC is that all rules are specific to a network device. Whenever a single controller presents multiple devices to the system, TC must be be configured independently for each. Combined with limited and expensive resources available for offloading the TC rules, this independence started to pose a problem.

The issue was solved recently by Jiří Pírko with the introduction of shared filter blocks [26]. The patch introduces a new, global, runtime entity of TC configuration – blocks. These fit in between qdiscs and filters. When no shared block is specified, a new private one is created for the qdisc, preserving backwards compatibility. In the other case, two or more qdiscs may share the defined policy.

The boilerplate needed to support TC offloading, however, got more complicated. Instead of handling events directly, the driver must register a callback on a newly-created block. The callback will then receive events from inside the block. Also, a future idea is that binding a block will replay all events, which is not yet supported. Instead, binding a block with any offloaded rule is forbidden.[5]

When it comes to hardware resource utilization, there is still an important unresolved problem. When the rules are to be compiled into table entries, it is usually necessary to fix the table dimensions in advance. For example the Flow

[5]Which is a bug confirmed by the author. Any configured rule should prevent the block from being bound.

Tables present in Mellanox ConnectX controller series must be allocated with a maximum number of rows in mind, and the Flow Groups must have fixed masks. Currently, the driver relies on grouping similar rules together and heuristics in table size allocation.

5. Proposed Subsystem

The current state of TC offloading is not ideal. It somehow works in practice, but certainly has some drawbacks. These drawbacks strenghten the motivation to explore other options. We would like to shortly review the most important problems of TC offloading:

Broken partial offloading
: When the user does not specify the `skip_sw` flags, the hardware can offload only a subset of rules. It is not guaranteed that the policy is preserved.

Flexibility
: While the flexibility of the TC subsystem is good for the user, it complicates offloading. The user works with a graph of rules which may run programs, but the hardware needs tables and simple actions.

Historic API
: The TC subsystem is complex and hard to understand. The code is burdened with almost 20 years worth of ad-hoc extensions. Its documentation is not up-to-date. Both the developers of the drivers and the users have hard time understanding the runtime structures.

Bad error reporting
: The only feedback from the subsystem with regard to offloading is a flag indicating whether a filter was offloaded or not. When it was not, the user has no way of knowing why.

With these drawbacks in mind, we would like to propose a subsystem that would remedy them. The subsystem is designed to be an offloadable representation of a match-action pipeline, which is present (in restricted forms) in the contemporary NICs. Along with avoiding the drawbacks of TC offloading, we focused on the following goals:

- Allow using as many packet-processing capabilities of modern NIC as possible.
- The subsystem must follow the Linux nature of being a hardware commoditizer, and allow a single configuration to run with any NIC. Of course, it can be offloaded only when the NIC is compatible, but the functionality should stay the same.
- When the user wants to program the controller directly, there are other solutions available. The subsystem should integrate well with the kernel datapath.
- Offloading the work of the subsystem should be as easy as possible. The API towards the drivers should be designed specifically for offloading.
- The hardware usually needs to allocate resources well in advance. Allow to restrict the resources in software to facilitate (or allow) the offload.
- The user should be able to understand the subsystem easily. The userspace utilities should interact with the subsystem through an understandable API.
- The offload must be restricted enough to preserve the defined policy, even when it is not possible to offload the setup completely.

- Recent controllers have good classification engines, but not so rich possibilities of modifying the packets. It should be possible to offload the classification independently.
- It is not possible to drop TC from the kernel. The subsystem must not interfere with TC if not used.
- There are multiple ways how to represent the policy for a particular scenario. Help the userspace with creating a setup which will be offloadable.
- When the subsystem is not used, it should not slow down the networking stack. We would like to avoid creating new hook as well.
- When the work cannot be offloaded, doing the work in software should not be extremely expensive when it comes to performance.

The key idea is to replace the TC role in the ACL, flow, and match-action pipeline offloading, without actually replacing TC itself. This proposal cannot be considered final and will be subject to discussion on the networking mailing lists before a patch will be created. Also, it is not meant as "all or nothing" – we present many ideas from which only a subset might be implemented in the end.

5.1 The Big Picture

The proposed subsystem is well separated – it is not a module of TC or netfilter, but rather a completely standalone entity. It has its own API against the userspace and drivers. This way, we can focus on creating an interface that is clean and straightforward to use.

The subsystem information base is stored per network namespace. In other words, entities created from inside a namespace are visible for all network devices inside that namespace. This way, we allow drivers to allocate resources once and share them between multiple `net_device`s of the same controller. This decision is supported by implementation of shared blocks in TC, as mentioned in Section 4.5.7.

The subsystem has a userspace configuration utility and a kernel module. The utility is a part of the `iproute2` package, and follows the same usage principles. The kernel module does the hard work in packet processing. It will be possible (and desirable) to control the subsystem through the netlink API, the utility is mainly for observability and manual testing purposes. The expected primary customer of the userspace API is a high-level software tool (e.g. intrusion detection system, SDN controller, ...) More about expected usage later.

The subsystem works with following entities: *tables*, *header fields*, *flows*, and *actions*. These terms are somewhat overloaded in the networking world, but in this chapter, we use them to reference the runtime entities of the proposed subsystem, unless specified otherwise.

From the top level view, *tables* form a directed graph. It is prohibited to create cycles, but it is not checked by the kernel module. There are several predefined types of tables, but their behavior is not different from the "generic" type from the point of view of the module. The purpose of type is to restrict the model in order to ease offloading for simpler devices.

Similarly, there is a preconfigured parser of several known *header fields* (MAC addresses, Ethertype, VLAN tag, IP addresses, TOS, ports, . . .). The parser tree

provides a standalone description of the packet header types and fields that can be extracted from them. This parser is extensible by generic *header fields* at runtime to support custom protocols. A detailed description follows in Section 5.4.

When a table is created, the set of header fields it uses is defined. This set is fixed and cannot be changed later. If the table is of one of the known types, the set of usable fields is restricted to a predefined subset. For example, a table of known table type "IP 5-tuple filter" could match on a subset of transport layer protocol, source and destination IP addresses, source and destination ports. If the table is of the generic type, it can use any subset of all the defined header fields.

Together with the header fields, a parser for the table has to be declared at creation time. We expect that for most of the time, the default Ethernet parser will be used.

We call the table entries *flows*. The system is expected to be continuously modified by inserting and deleting flows from tables. Every flow has an individual chain of *actions* assigned.

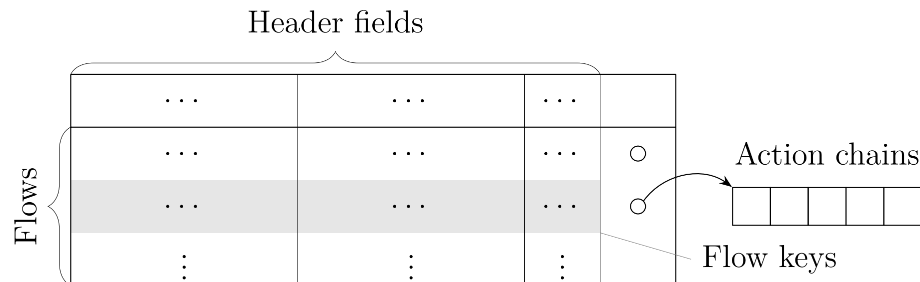

Figure 5.1: The entities in a table.

Now we can finally explain what the main processing looks like: Say that the subsystem is called to process a packet using table T. The header fields that T uses are extracted from the packet using the parser of T, and are joined to create a *flow key*. The flow key is used to search the table and identify a *flow*. When the flow is identified, a sequence of *actions* is executed. If *next table* was set while T was processed, the processing repeats with the next table.

To prevent infinite loops, the process stops after a constant number of iterations. This limit is rather high and reaching it emits a warning in a rate-limited manner. To prevent malformed packets from reaching the system, the packet trapped in such loop is dropped.

5.1.1 Matching Modes

Every table uses one of the predefined matching modes on the header fields. Those we propose are inspired by options available in the hardware pipelines:

Exact All used fields must match exactly.

Hashed The flow key is hashed and then only hashes are compared.

Mask-value Emulate a TCAM search. The flow key is compared only on bits specified by the flow.

Range Every field is checked to be inside an interval.

Longest-prefix match A flow with the longest matching prefix is selected.

In the hashed mode, we do not specify the hash function to be used. The mode serves as a hint for the hardware that it can afford collisions. In fact, the exact mode can be implemented with hash tables as well.

5.1.2 Actions

Apart from action chains assigned to individual flows, the table contains three more chains. The first is the *default* chain, which is executed when no particular flow is matched. The other two are a *pre-* and *post-*chain, which are executed every time. The execution starts with the pre-chain, follows with the flow chain (or default chain) and finishes with the post-chain.

Any action chain can be changed at runtime. Modifying a chain can result in change of table offload status – either it can enable offloading of the previously not-offloaded table or vice versa.

An action chain is composed of primitive actions. The set of available primitive actions is specified by the subsystem and generally is not expected to be extendable by kernel modules. By design, the action system is much simpler than that of TC.

Primitive actions can be parameterized. In terms of implementation, we propose a simple union containing possible argument types to keep things simple.

To create a common understanding of action purpose and flexibility, an incomplete list of primitive actions follows:

`drop`
: Drop the packet. Immediately stops processing.

`set next table <T>`
: After the processing of this table finishes, table `<T>` is consulted next. Multiple usages of this action overwrite each other.

`stop`
: Stop processing this table immediately.

`set field <field> <value>`
: Modify the header field `<field>` to contain `<value>`.

`copy value <field1> <field2>`
: Copy the value from `<field1>` to `<field2>`.

`set queue <Q>`
: On ingress, set the receive queue to which the packet will be enqueued (makes sense only in hardware). On egress, set the transmit queue (makes sense only in software).

`mirror to port <P>`
: Send a copy of the packet to egress on port `<P>`. As the port is device-dependent, using such action disables offloading of the flow for all other devices.

Even though the action system is simplified, we could extend it with conditionals and create a really flexible system. On the other hand, the purpose of

the subsystem is to create better offloading opportunities, and more complicated packet-processing code should be implemented with different technologies (XDP, custom firmware, ...).

To support more advanced features of modern NICs, we generally prefer adding a slightly complex primitive action than emulating it with several primitive ones. The older or less powerful controllers are not able to offload more complex chains anyway, and more specialized actions will result in less complex drivers. A reasonable overlap in action functionalities is acceptable.

Let us extend the example of the “IP 5-tuple filter”. Such filter can be defined as a table with mask-value match mode to allow wildcard rules. The default action is empty, passing the packet through. Flows inserted to the table are created with the action chain containing only the `drop` action. The same table could be created as a whitelist filter by using the `drop` action in the default chain and empty chains for selected flows.

5.1.3 Offloading

To allow NIC drivers to offload the subsystem work, a separate API is provided. Through this API, the driver registers for updates on a table. Those will be delivered by invoking callbacks specified in a structure of operations. Events will notify the driver of inserted flows and other runtime modifications. At the time of registration to a table, the current state is completely replayed (not in original order). This asynchronous approach is needed to avoid locking the table for driver introspection.

There are two more details which we need to mention before we explain the offload in detail. First, tables can be created with a maximum size specified. The driver can use this property to allocate resources in advance. However, the size can be changed at runtime, but the resize action can result in the offload being stopped.

Second, action chains are stored separately from flow matching. When a flow is matched, an action chain ID is obtained. The chain ID is then used to do another lookup in a hash table of action chains. This allows for two major optimizations: identical chains can be merged into one and classification can be offloaded separately from action execution.

When a table is created, the driver can check for unknown fields or impossible combination of fields, and map the table into the controller pipeline. If the driver is rather simple one, it can just map known table types. If on the other hand the controller is fully programmable, it knows all the information to configure it.

Then, the driver can check the table-wide chains for unknown action primitives or impossible action chains. Because of the purpose of the system, the driver can be rather strict – for example, filter tables should contain chains comprised of either a `drop` action or nothing.

Let us start with the ingress path. There already is a flag `tc_skip_classify` in the `sk_buff` structure which indicates that the packet was already classified (and acted upon) by the hardware. We plan to use this bit (see section 5.3 for better insight into this decision) for the same purpose.

The driver configures the card to offload any prefix of the table graph. If the graph can be offloaded completely, it can just mark the packet to be skipped

and deliver the packet to the networking stack. In the other case, the graph was processed just partially and needs to be finished in software. By vendor-specific means, the driver should know in which phase the processing was interrupted, and therefore know where to continue. For the purpose of partial offloading, the API of the subsystem exposes the executor state structure, which should be created accordingly by the driver. Then, the driver should call the executor of the subsystem to finish the processing.

The egress part is a bit trickier. The driver can instruct the subsystem to completely avoid processing the packet in the software pipeline. This is configurable per `net_device`. An unprocessed packet is then received by the driver. Again, if the table graph can be offloaded completely, no action needs to be done in software and the packet can be given directly to the hardware. Conversely, when the graph is not offloadable as a whole, the driver must run the executor in software until it can be sure that all reachable tables are offloaded. Then it can hand out the packet to the controller.

The subsystem allows partial offloading of a single table. As already said, the controller can perform only classification, passing the action chain ID to the software out of band. However, the subsystem in principle allows to offload a subset of flows from a table, provided that the driver ensures the policy is preserved. For example, the driver can evaluate that the "IP 5-tuple blacklist filter" is idempotent for missed packets, thus can be performed both in the hardware and the software. Then, offloading a subset of rules to the hardware serves as an optimization.

However, such usage is discouraged, because the driver would have to implement the algorithm to select offloaded flows, moving the complexity to individual drivers. The encouraged solution is to offload tables of limited size only, forcing the user to implement the rule aging in the upper layers.

5.2 Acting as an OpenFlow Backend

A reader familiar with OpenFlow can notice that the subsystem can serve as a backend to implement OpenFlow-compatible software switch. Indeed, this was one of the design goals of the subsystem. If the proposed subsystem was implemented, one would just need to write a daemon for communication with the SDN controller and translate it into calls of the proposed subsystem API.

5.3 Compatibility

So far, we did not explain when and how is the subsystem going to be inserted into the current software pipeline. An obvious answer would be to add another hook in there, as close to the hardware as possible (probably under TC). We decided that it is not necessary, and we would rather avoid doing it.

Instead, we propose implementing a classifier for TC. In the architecture of TC, classifiers are in charge of executing actions, therefore it shall be acceptable that our classifier would act on the packet on its own. As for the TC actions, we can prohibit attaching them to our filter.

The model we propose is similar to the `bpf` TC classifier, which is already present in the upstream. Also, eBPF programs can modify the packets as well, so the separated action model should not pose a problem for the community.

To further support hardware offloading, we would extend the recently-added block mechanism in TC with a special type of block. The proposed subsystem would expose blocks of this type to the TC, and userspace could attach them to `clsact` ingress/egress hooks. As only one block can be hooked there, the driver could be sure that the proposed subsystem pipeline is the only processing that happens.

Summed up, what we propose is essentially replacing the `clsact` hooks in a non-intrusive way. This has the advantage of exposing our own userspace API separate from TC, while reusing as much in-kernel infrastructure as possible without sacrificing any of the goals. For example, we can reuse the `ndo_setup_tc` callback up to the point where blocks are examined.

As the subsystem is very generic, its functionality naturally overlaps with some more specific features of the Linux kernel. For example, it can theoretically be used instead of RFS. It is probably not a good idea to do so, as the software path would be almost certainly slower and the offloaded table more generic than RFS, making its offload in fact more complicated. However, it is up to the driver if it utilizes a part of its hardware pipeline to offload RFS or the proposed subsystem.

For a concrete example, take the Intel Flow Director filters. They can be used to offload RFS, Ethtool Flow Classification, or the subsystem we propose. The driver can either make a fixed decision, switching between these offloads during compilation, let the user make the choice using driver-specific configuration tools, or use some heuristics to select the most beneficial offload automatically.

One could argue that for generic, low-level packet processing the kernel already contains eXpress Data Path (XDP). However, offloading classification is considerably simpler than running an arbitrary BPF program.

5.4 Configurable Parser

This whole section is proposed as an optional extension. It can be implemented separately and afterwards.

Apart from using the kernel flow-dissector, we would implement our own packet parser. The parser graph would be represented as a runtime entity, and thus could be modified from userspace. The parser graph could be extended to support new header fields or even new protocol headers.

The data structure representing a header field would carry a `type`. Every well-known header field would have its own type, and those types would be still extracted by the flow dissector or taken from the `sk_buff` structure directly. Only the fields whose type is "generic" would require parsing the packet to extract their value.

The parser graph is composed of nodes representing various protocols. We call the nodes *headers*. Every header field is contained in exactly one header. Headers describe how the parser walks the packets and identifies protocols. Header fields describe how to extract a field value from a header instance.

Packet parsing would be similar to table processing. The parser starts with a header defined by the link layer protocol – for us, Ethernet. If there is a header

field which needs to be extracted from the Ethernet header, it is extracted. Then, a mask-value match is performed on predefined fields to determine the next header. If no next header matches, the parsing is over.

The predefined parser would be fixed, and it would not be possible to overwrite the default rules and header fields. It would be however possible to extend the parser with new fields and headers hooked up to places where the processing would end previously. This is necessary to avoid changing the definition of well-known fields.

If there are no unknown fields allowed in a table, the driver does not need to worry about the configurable parser. If there are some, the driver can have a look on the parser tree definition. In the simple case, the field would be just a previously-unknown field of a known header. Several controllers feature matching on a whole header or arbitrary part of it, enabling offload of those fields. In the most complicated case, new parser states would be defined. If the controller features a programmable parser, the driver can can program it accordingly using the information from the parser tree.

5.5 Introspection

So far, the system was completely independent of the controllers present in the system. In the real world, the user (or userspace software) would probably want to know how to arrange the tables to make them most effective. We are proposing introspection capabilities, which the userspace could utilize to optimize for offloading.

The first challenge to be solved is that devices usually have only partially programmable pipeline. For this purpose, the driver would export a graph of fixed tables along with *extension points* – places where generic tables could be created.

Next, we have to describe these fixed tables. Throughout the chapter, we complicated matters by defining things both generic and static (well-known) – tables, parser states (headers), header fields. Now it comes in handy – the driver can use these values to describe known fields in a compact way. For globally unknown but fixed entities, the driver can make use of the generic types.

Next, we have to describe these fixed tables. In common cases, the driver might use the well-known table types. Where the table is fixed but not well-known, it can be described by the generic table type, for which the usable header fields, maximum size, etc. are defined (if applicable).

The hardest difficulty is to describe how much flexibility the user has in defining generic entities (configurable parser, generic tables in extension points). For headers and header fields, we think that describing the configurability is of no use, because if the use case requires matching on generic field or whole new protocol header, it probably cannot be avoided. On the other hand, a description of the flexibility permitted in table definition could be very useful. However, we think that we just cannot cover arbitrary hardware (contemporary or future one) accurately enough, unless the description itself is Turing-complete. But then it would just be too complex for the purpose.

Instead, we suggest to go similar way the TC offload used – enforce table offload by a flag. Whenever a table cannot be offloaded by the device, the binding

of its block should fail. Similarly, when a flow inserted to such a table cannot be offloaded, or the table is resized beyond hardware limits, the action should fail immediately.

Also, the userspace has the option of querying the device model name and version, and looking up the available features in its own database. Such database and the model it describes might be much more flexible than that in kernel, which needs to be backwards compatible forever.

Note that while mixing offloadable and non-offloadable rules in TC can result in unexpected behavior, the same situation does not happen here. The driver is forced to offload a complete prefix (or suffix) of the pipeline – marking table as forcefully offloaded results in transitively enforced offloading of other tables. Say that table T is marked with `skip_sw` flag. When bound to ingress, the driver must offload all tables from which T is reachable. When bound to egress, all tables reachable from T must be offloaded.

To support drivers in these operations, the table graph would be maintained explicitly with lists of neighbors in both directions. This also allows the userspace to look for cycles faster.

5.6 Acting as a P4 Backend

As the subsystem behavior closely follows the current hardware, we can use the existing tools for programming the hardware pipelines. The P4 language [4] serves exactly this purpose. It allows a system administrator to define the behavior of a pipeline in a restricted imperative language. The program can then be compiled for a specific hardware pipeline. The compilers can already handle situations where pipelines are partially programmable.

We allow for creation of a backend for such compiler. The compiler would translate a P4 program to a series of commands (or full state description) that would emulate the desired behavior using our subsystem. The compiler could even introspect the current hardware and optimize the program for it, resulting in a pipeline which is well-offloaded.

5.7 Simplifying Overlay

We admit that the mechanism might be overly generic and flexible. It is however necessary to cover all the different hardware designs and prepare for the future. The complexity of the system does not prevent us from wrapping it in a simple interface – it would not be possible the other way around.

For example, a “library” could be created for drivers. The library could serve drivers which are completely inflexible in terms of generic tables and parsers. We could select several well-known table types and allow the driver to register callbacks for flow insertion/deletion on these. The boilerplate for drivers would be reduced to a bare minimum of implementing operations in the spirit of “filter this particular VLAN ID on this port” or “insert an entry into FDB”.

Similarly, we could create a library for userspace. The user of the library would be abstracted from the complexity of the system, and would work with

well-known tables only. Those tables could be provided with a sensible interface, understanding how to parse and display addresses, ports, constants and so on.

5.8 Performance

If we look at the system as a whole, we can say it is an emulator of a programmable hardware pipeline. As such, we do not expect it to have miraculous performance when run in software. Yet, if we compare it with current state of the art, we expect it to be similar to TC with `flower` filters installed. It probably would not be faster than a hand-tuned `u32` classifier tree, which can be optimized much further. However, such optimized solution is unlikely to be offloaded by any current driver.

Furthermore, we have the advantage of a more restricted behavior, and therefore we could utilize some clever algorithms to speed up matching of mask-value rules. The tables have the interface of a dictionary, unlike the TC rules, which are a programming language. A dictionary could be optimized into a decision tree automatically, while TC rules must be evaluated strictly in order to preserve correctness.

Besides, we expect to solve an existing performance problem – rule insertion rate. Currently, inserting a rule into TC requires the `rtnl_mutex` to be taken. There are ongoing efforts to remove it, but the parallelization of previously serial code is a notoriously hard problem. By contrast, inserting a flow into a hash-table or a tree could be implemented with fine-grained locking or even lockless data structures.

For use cases with maximum required software performance, neither our subsystem nor TC is ideal. XDP serves those purposes much better. When the features of the Linux networking stack are not required, it might be even better to employ a kernel-bypassing solution like DPDK.

Conclusion

We succeeded in our goal to design a mechanism that would allow the user to define a policy to process packets in the kernel, while allowing to offload the policy into compatible NICs. Rather than following the initial plan to pick a few common features and create a specific mechanism for them, we proposed a generic mechanism to offload the match-action processing.

We did so after carefully studying the pipeline of five high-end NICs. The subsystem we proposed is descriptive enough to represent the majority of classification and packet-modification features these NICs have while being restricted enough to be offloaded easily.

Unfortunately, we have not succeeded in getting access to confidential documents to support the research. Multiple vendors refused to give us information that is not already public. Therefore, the thesis builds upon public information only.

The proposed subsystem could replace the TC offload. It is better suited to be offloaded to hardware, and thanks to a different approach it resolves some serious problems of the TC offload.

It is not the first work of its kind. Most notably, John Fastabend proposed the Flow API [9], which might seem very similar. The Flow API, however, serves to precisely describe a fixed pipeline of the controller and to allow the userspace to program it directly, bypassing the kernel. In contrast, our subsystem starts with doing the work in the kernel and then offering the drivers to offload it.

Another project that is somewhat similar is the support for Flowtables in Netfilter, implemented by Pablo Neira Ayuso [2]. The Flowtables are limited to offloading forwarding, and take a "reversed" approach – it is the user who decides which particular rules should be offloaded. The drivers create the tables and the user fills them.

To support our design, we created a proof-of-concept implementation, which is attached to the electronic version of the thesis. It is also available on Github[1]. The documentation of the implementation can be found in Appendix A. The demonstration shows that the subsystem can be created, can be used to process packets and that the drivers can offload its work.

While studying the source codes, we have discovered two bugs in the Linux kernel. Both of them have been reported to the respective maintainers. Unfortunately, only one of them replied and confirmed the bug. As for the other bug, we are currently testing the patch and plan to submit it soon.

In the future, we plan to send the proposal of the subsystem to the Linux NetDev community.

[1] `https://github.com/Aearsis/mat`

Bibliography

[1] Almesberger, Werner, Jamal Hadi Salim, and Alexander Kuznetsov. "Differentiated services on Linux". In: *Global Telecommunications Conference, 1999. GLOBECOM '99*. Vol. 1B. 1999. DOI: 10.1109/GLOCOM.1999.830189.

[2] Ayuso, Pablo Neira. *Flow offload infrastructure.* URL: https://lwn.net/Articles/738214/ (visited on 2018-05-01).

[3] Borkmann, Daniel. *net, sched: add clsact qdisc*. 2016-01-07. URL: https://lwn.net/Articles/671458/ (visited on 2018-04-22).

[4] Bosshart, Pat et al. "P4: Programming Protocol-independent Packet Processors". In: *SIGCOMM Comput. Commun. Rev.* 44.3 (2014-07), pp. 87–95. ISSN: 0146-4833. DOI: 10.1145/2656877.2656890.

[5] Chelsio Communications. *Chelsio Unified Wire for Linux*. Version 1.3.9.

[6] Chelsio Communications. *The Chelsio Terminator 6 ASIC.*

[7] Cree, Edward. *Local Checksum Offload.* 2016-01-07. URL: https://lwn.net/Articles/671457/ (visited on 2018-04-23).

[8] Deering, Steve and Robert M. Hinden. *RFC 2460: Internet Protocol, Version 6 (IPv6) Specification.* 1998-12. DOI: 10.17487/RFC2460.

[9] Fastabend, John. *A Flow API for Linux Hardware Devices.* URL: http://people.netfilter.org/pablo/netdev0.1/papers/A-Flow-API-for-Linux-Hardware-Devices.pdf (visited on 2018-05-01).

[10] Fastabend, John. *net: implement mechanism for HW based QOS*. 2010-12-17. URL: https://lwn.net/Articles/420509/ (visited on 2018-04-22).

[11] Gerlitz, Or. *Introducing ConnectX-4 Ethernet SRIOV*. 2015-11-23. URL: https://lwn.net/Articles/666180/ (visited on 2018-04-07).

[12] Gross, Jesse, Ilango Ganga, and T. Sridhar. *Geneve: Generic Network Virtualization Encapsulation.* Internet-Draft. URL: https://datatracker.ietf.org/doc/html/draft-ietf-nvo3-geneve-06.

[13] Herbert, Tom. *rfs: Receive Flow Steering.* 2010-04-01. URL: https://lwn.net/Articles/381955/ (visited on 2018-04-19).

[14] "IEEE Standard for Ethernet". In: *IEEE Std 802.3-2015 (Revision of IEEE Std 802.3-2012)* (2016-03). DOI: 10.1109/IEEESTD.2016.7428776.

[15] Intel Corporation. *Intel 82599 10 GbE Controller Datasheet.* Version 3.3.

[16] Intel Corporation. *Intel Ethernet Controller XL710/XXV710/XL710 Datasheet.* Version 3.5.

[17] Linux Foundation. *TCP Offload Engine.* URL: https://wiki.linuxfoundation.org/networking/toe (visited on 2018-03-23).

[18] McCanne, Steven and Van Jacobson. "The BSD Packet Filter: A New Architecture for User-level Packet Capture". In: (1992-12-19).

[19] Mellanox Technologies. *ConnectX-5 EN IC: Product Brief.* Version 1.2.

[20] Mellanox Technologies. *Mellanox Adapters Programmer's Reference Manual*. Version 0.40.

[21] Microsoft. *Network Driver Design Guide*. 2017-04-11.
URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/network/ (visited on 2018-03-23).

[22] Miller, David. *The net-next tree of Linux kernel*.
URL: https://git.kernel.org/pub/scm/linux/kernel/git/davem/net-next.git/ (visited on 2018-05-03).

[23] Netronome Systems. *NFP-4000 Theory of Operation*.
URL: https://www.netronome.com/media/documents/WP_NFP4000_TOO.pdf (visited on 2018-03-24).

[24] Netronome Systems. *Programming Netronome Agilio® SmartNICs*.
URL: https://www.netronome.com/media/documents/WP_NFP_Programming_Model.pdf (visited on 2018-03-24).

[25] Open Networking Foundation. *OpenFlow Switch Specification*. Ver. 1.5.1. 2015-03-26.
URL: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf (visited on 2018-05-07).

[26] Pírko, Jiří. *net: sched: allow qdiscs to share filter block instances*. 2018-01-06.
URL: https://lwn.net/Articles/743391/ (visited on 2018-04-22).

[27] Postel, Jon. *RFC 791: Internet Protocol*. 1981-09. DOI: 10.17487/RFC0791.

[28] Rosen, Rami. *Linux Kernel Networking: Implementation and Theory*. Apress, 2014. ISBN: 978-1-4302-6196-4. DOI: 10.1007/978-1-4302-6197-1.

[29] Stuart Wray. *The Joy of Micro-C*.
URL: https://open-nfp.org/m/documents/the-joy-of-micro-c_fcjSfra.pdf (visited on 2018-03-24).

[30] Torvalds, Linus. *Linux kernel source tree*.
URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?h=v4.16 (visited on 2018-04-02).

[31] Touch, Dr. Joseph D. *RFC 6864: Updated Specification of the IPv4 ID Field*. 2013-02. DOI: 10.17487/RFC6864.

List of Figures

4.1 Example qdisc tree 43

5.1 Subsystem entities 51

A.1 Components of the demonstrator 68
A.2 Modules of the demonstrator 69
A.3 An example of packet parsing 71

Acronyms

ABI Application Binary Interface. 70
ACL Access Control Lists. 15, 50
API Application Programming Interface. 15, 34, 49, 50, 53–55
ARM Advanced RISC Machine. 29
ASIC Application Specific Integrated Circuit. 12, 20, 26
BPF Berkeley Packet Filter. 40, 41, 55
CAM Content Addressable Memory. 12, 13
CPU Central Processing Unit. 3, 11, 38–40
CRC Cyclic Redundancy Check. 8, 20, 34
DCB Data Center Bridging. 18–20
DCCP Datagram Congestion Control Protocol. 8
DMA Direct Memory Access. 31, 35
DoS Denial of Service. 15, 40
DPDK Data Plane Development Kit. 3, 4, 58
E-Switch Ethernet Switch. 24, 25
eBPF Extended Berkeley Packet Filter. 55
FCoE Fibre Channel over Ethernet. 18–20, 27, 34
FIFO First In, First Out. 42
FPC Flow Processing Core. 29
GPU Graphics Processing Unit. 29
GRE Generic Routing Encapsulation. 8, 14, 21, 24, 26, 65
GRO Generic Receive Offload. 35, 36
GSO Generic Segmentation Offload. 34, 36, 37
IEEE Institute of Electrical and Electronics Engineers. 18
IP Internet Protocol. 4, 5, 8–10, 12, 17–19, 21, 22, 27, 34, 35, 39, 50, 51
IPv4 Internet Protocol, version 4 [27]. 4–6, 8, 10, 12, 19, 24–26, 34, 37, 43
IPv6 Internet Protocol, version 6 [8]. 4, 8, 12, 18, 24, 25, 33, 34, 43
iSCSI Internet Small Computer System Interface. 27
iWARP Internet Wide-Area RDMA Protocol. 27
JIT Just In Time (compilation). 41
L4 Transport Layer in the TCP/IP model. 27, 39
LLDP Link-Layer Discovery Protocol. 18
LRO Large Receive Offload. 9, 10, 17, 18, 21, 35, 36
LSO Large Send Offload. 9, 21, 25, 26
MAC Media Access Control. 12, 14, 15, 19–22, 24, 26–28, 35, 50, 69–71
MAUI Media Attachment Unit Interface. 21
MPFS Multi Physical Function Switch. 24
MSS Maximum Segment Size. 37
MTU Minimum Transmission Unit. 9
NAPI New API (of the networking stack in Linux kernel). 35, 36, 38
NAT Network Address Translation. 27
NDIS Network Driver Interface Specification. 7, 31
NIC Network Interface Controller. 3, 6–14, 16, 17, 26, 27, 29, 31–34, 36–39, 49, 53, 59, 67, 68, 70, 72
NVGRE Network Virtualization using GRE. 21
NVM Non-Volatile Memory. 21
NVMe-oF Non-Volatile Memory over Fabric. 27
P4 Programming Protocol-Independent Packet Processors. 29, 57
PCI Peripheral Component Interconnect. 24, 65
PCIe PCI Express. 13, 17
PCP Priority Code Point. 19
PDU Protocol Data Unit. 8, 10
PHY Physical layer. 21, 23
PRM Programmer's Reference Manual. 23

qdisc Queue Discipline (in TC). 41–46, 63

QoS Quality of Service. 10, 44, 45

RAM Random Access Memory. 12, 13

RDMA Remote Direct Memory Access. 27, 65

RFS Receive Flow Steering. 38, 39, 55

RISC Reduced Instruction Set Computing. 65

RPS Receive Packet Steering. 38–40

RSC Receive Segment Coalescing. 18

RSS Receive-Side Scaling. 1, 11, 18, 19, 25, 27, 38, 39

SA Security Association. 20

SCTP Stream Control Transmission Protocol. 8, 17, 34

SDK Software Development Kit. 29

SDN Software Defined Networking. 13, 14, 50, 54

SerDes Serializer-Deserializer. 23

SMB Server Message Block. 27

SOHO Small Office / Home Office. 14

SR-IOV Single Root IO Virtualization. 13, 14, 17, 19–24, 26

TC Traffic Control (subsystem in Linux). 3, 6, 41–46, 49, 50, 52, 54–59, 66, 67, 72

TCAM Ternary Content Addressable Memory. 13, 24, 27, 51, 70

TCP Transmission Control Protocol. 1, 8–12, 17–19, 21, 24, 25, 27, 34–37, 42, 66

TIR Transport Interface Receive. 25, 26

TIS Transport Interface Send. 25, 26

TLS Transport-Layer Security. 27

TOE TCP Offload Engine. 10, 27, 37

TOS Type Of Service. 12, 42, 50

TSO TCP Segmentation Offload. 9, 34, 36

TTL Time To Live. 72

UDP User Datagram Protocol. 5, 6, 8, 9, 11, 17, 18, 21, 24–27, 34, 35

VLAN Virtual Local Area Network. 1, 12, 14–16, 19, 22–25, 27, 33, 39, 44, 50, 57, 70, 71

VM Virtual Machine. 22

VMDq Virtual Machine Device Queues. 19, 22

VSI Virtual Station Interface. 22, 23

VXLAN Virtual Extensible LAN. 9, 14, 21, 24, 26, 68

XDP eXpress Data Path. 3, 40, 41, 53, 55, 58

XPS Transmit Packet Steering. 40

A. The Demonstrator

The attached source code contains an implementation of the subsystem under a working title Match-Action Tables, MAT for short. In this appendix, we would like to document the source code from a high-level point of view. We do not discuss implementation details, as they are documented in the source code. Also, we do not discuss the functionality itself, as it closely follows the model presented in Section 5.

You will not find an implementation of a Linux kernel module in the archive. Instead, we created an environment where the implementation looks like if it was written as a kernel module – we could say that the MAT subsystem is written against a *mocked* kernel. Most importantly, all of the code actually runs in the userspace, even though it simulates a code being run in both the kernel mode and the userspace.

A.1 Usage

The source code is written in the C language and follows the GNU89 standard. No libraries are required to compile it. The code still uses some Linux headers for convenience, we do not expect it to be compilable on other platforms. Due to the target audience, we do not consider it a problem.

The archive contains a `Makefile` that should compile all the sources by running `make`. If the compilation was successful, you can run any of the tests from the `tests` directory. Also, you can run the `test.sh` script that runs all the tests and compares their outputs to the expected ones.

The tests usually do some configuration and simulate receiving packets in between individual configuration steps. As the subsystem produces debug messages to the output, you can see what the individual components do. The expected behavior is usually printed before running the steps and documented more extensively in the source code.

The sources of the tests are more interesting than their outputs. They show the expected userspace interface and the configuration of the subsystem. To simulate packet reception, the test code acts both as the userspace and the NIC driver, as shown on Figure A.1.

A.2 Implementation Overview

At the top level, we can split the implementation into three components. First, there is the kernel mock. It defines structures like `sk_buff` or `net_device`, but also contains bits extracted from the internal API of TC. Also, we extracted the implementation of linked lists and several general-purpose macros. To avoid copying the `idr` structure[1] implementation, we simply used linearly increasing values and a very simple hashtable/array. Also, we implement the kernel dynamic memory allocation routines `kzalloc` and `kfree` using the `libc` `calloc` and `free`, just to make the code look like a kernel source code. In contrast, we decided not

[1] Radix tree used to allocate and map integer identifiers to objects.

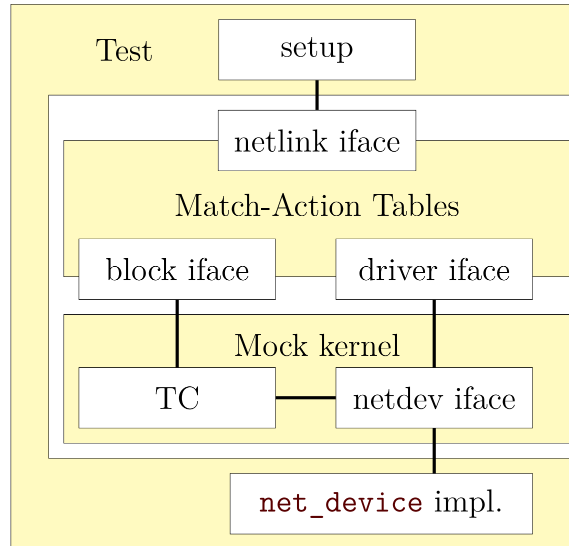

Figure A.1: The components of the demonstration implementation.

to implement `printk` to make it obvious that the prints are for demonstration purposes only.

Second, there is the implementation of the MAT subsystem. We will describe this part in detail in the rest of the appendix. Together with the kernel mock, the implementation is linked to a single archive `libmat.a` to represent "the kernel".

Finally, the archive contains a few test scenarios to demonstrate how the subsystem is expected to be used. As already mentioned, the test scenarios interact with the kernel from both userspace and hardware ends. Let us have a look at them first.

A.2.1 Tests

There are five tests included:

`exact`, `hash`, and `tcam`
: These configure a table of the given type, set the default action chain and insert a rule to drop matching packets. The tests demonstrate how the tables are configured and test the software implementation of the subsystem.

`simple-nic`
: This test demonstrates the driver interface and the hardware offload mechanism. The emulated NIC contains a blacklist 5-tuple filter in its pipeline. The test configures a table which matches on IP addresses and simulates receiving packets. It demonstrates how the ingress processing is moved to the NIC.

`parser`
: This test extends the configurable parser with the VXLAN header. The parser tree is printed before and after to demonstrate the extension. Then, a simple table is created to show how the driver can read the parser state to offload it.

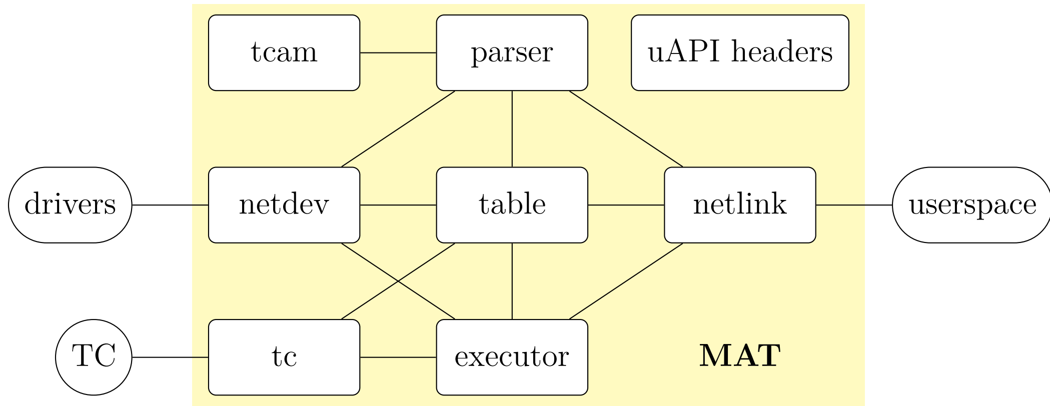

Figure A.2: The modules of our implementation. Highlighted is the scope of the MAT subsystem. Relations show communicating modules.

`multi-table`
: A chain of tables demonstrating a more complex pipeline is tested. The tables represent a non-optimal firewall and unicast MAC filter.

Please note that having multiple demonstrations of the offload mechanism would require to implement all the functionality to simulate the hardware processing, duplicating the functionality of the subsystem in the tests. As the principles stay the same for other types of tables, we believe one demonstration is enough to show the principles of offloading.

A.2.2 Match-Action Tables

The implementation of the subsystem can be split into several modules, as seen on Figure A.2. Not all dependencies are displayed.

The implementation is rather simplified. It is not meant to be production-ready, but rather serve as a demonstration of thoughts and principles. It should not even be used as a base for the real implementation, because some of the things we ignore are hard to add as an afterthought. Most notably, we purposefully ignore:

Multithreading and synchronization
: As synchronization in the kernel is vastly different from that of userspace, we decided to omit it completely. This alone should be a reason to write the real implementation from scratch.

Deconfiguration
: To deliver the ideas, it is necessary to support creating the configuration. However, there is a lot of bookkeeping code to support removing or changing the configured entities. We decided to keep things simple for the demonstration.

  Also, when things are not removed, they are usually never deallocated. We believe that the destruction routines are easy to imagine.

Error paths
: As the implementation focuses on the ideas, we consider handling error

paths an unnecessary noise. Also, as there are no routines to handle removing things, there is no sensible way how to handle errors.

Netlink method calling
: Where the netlink interface is just a plain call to a function with fixed arguments, we decided to just call the function directly. The netlink protocol is designed to be very flexible and binary compatible, but not particularly readable. Hiding the actual interface into netlink messages would introduce unnecessary clutter.

However, we did not simplify the important things. We strictly separate the kernel and userspace memory areas. Data structures can be shared only from the userspace to the kernel, not the other way around. Such structures are declared in a shared header file, which is separate from the implementation. As already said, the header file contains also the definitions of functions, which can be easily transformed into simple calls over the netlink interface.

The `netdev` and `tc` are simple modules that serve as brokers to communicate with other parts of the kernel. In the real-world implementation, these would probably get thicker to better separate the implementation from the drivers, supporting the stability of the kernel ABI.

Parser

The `parser` module keeps the representation of the packet parser. The parser is defined completely as a runtime data structure to allow extensions. The module defines two important structures: `mat_parser` and `mat_header_field`. Instances of both are identified by globally unique indices.

In the following paragraphs, we use the term "parser" for both the instances of the `mat_parser` structure and the parser tree. In fact, the parser tree is nothing more than the root parser instance for the Ethernet header.

An instance of `mat_parser` is created for every protocol defined. The parser is used to identify fields from a packet of the corresponding protocol and to determine the parser for the next-layer header.

The `mat_header_field` structure instances correspond to individual header fields. The structure contains everything that is needed to extract and interpret the field value from the byte stream. In case the field is expected to be used to match on, it is assigned to a parser.

Every parser (except for the innermost protocols) has two fields by default – `nexthdr` and `hdrsize`. The former identifies the field used to determine the next-layer parser, the latter determines the distance to the next header in bytes.

To define which parser will be used to parse the next header, a mask-value match is performed. The rules are stored in an instance of `mat_tcam` structure, provided by the `tcam` module. The structure simulates a TCAM.

The preconfigured parser identifies some basic fields and shows how to solve several challenges. As an example of a challenge, the *Ethertype* field is identified by the NICs as the Ethertype of the last VLAN tag, or the MAC header directly if no VLAN tags are present. As such header field can be attached to at most one parser, we define the Ethertype as a standalone parser and make the Ethernet

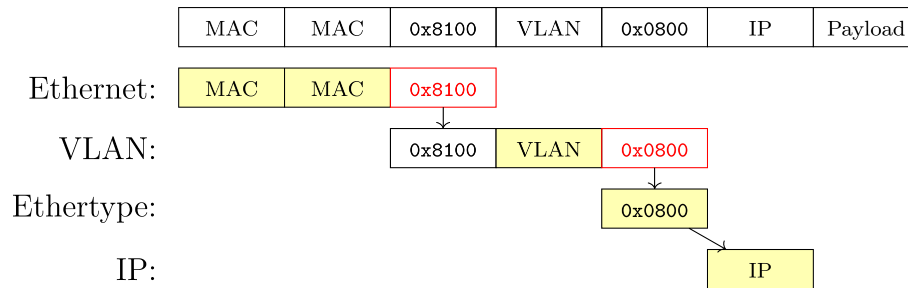

Figure A.3: Parsing a VLAN-tagged packet with the preconfigured parser. Parser states and their scope are in horizontal layers. Look-ahead fields shown in red, identified fields in yellow. Size not to scale.

and VLAN parsers "look ahead", out of the area that is skipped by parsing the corresponding header. For illustration, have a look at Figure A.3.

It is important to note that the preconfigured parser tree should never be actually used to extract fields. The values of well-known fields are already extracted by the kernel flow dissector and thus present in the `sk_buff`. However, the definition needs to be precise so as not to break the offload to the programmable parsers.

Table

The `table` module stores the configured tables. There are two main structures defined: `mat_table` and `mat_flow_key`.

The purpose of the `mat_table` structure is to keep all the information about a table. Even though it is one of the largest structures in the codebase, there is nothing unexpected.

The `mat_flow_key` structure is a pure data carrier for flow keys. Both its size and interpretation of content is determined by the table for which the flow key is created. Therefore, the flow key is always associated with a concrete table instance, even though it does not carry a pointer to the table.

When a table is created, a mapping of fields to the flow key bits is computed. This part solves Bin packing problem to distribute variable-sized fields to the least number of 64-bit *parts* of the flow key. The number of parts is then recorded in the table.

As the flow key interpretation depends on the table, it is the `table` module that manipulates it, including filling the flow key with the data parsed from the packet.

Every table holds its flows and action chains separately. Action chains are stored in a hash table, in which the chains are identified by a unique 64-bit number, *Action ID*. In the future, the ID could be computed from the content of the chain, allowing to merge identical chains into one.

Determining the Action ID varies with the table type. Tables of the *Hash* type use the flow hash directly, and therefore do not need to store any additional data. Other types use different data structures following their purpose.

Executor

The software functionality of the subsystem is implemented in the `executor` module. That includes both executing the action chains and taking the packet through the pipeline of tables.

To allow fine-grained control by the device drivers, which might need to resume the processing in software after offloading it partially to hardware, the module exports its control structure, `mat_executor`. The structure contains the execution state as well as a few variables that control the execution.

To start the executor from the beginning, the exported TC filter initializes the structure and calls `mat_executor_run`. The executor then performs all the necessary steps, until the final verdict is returned.

When the driver needs to start the execution from a different point (e.g. when the hardware classified the packet, but is not able to execute the actions), the driver initializes the `mat_executor` structure, fills the known information and uses the structure to resume the execution in software.

The `mat_executor` structure holds a TTL number. With every step executed, the TTL is decreased. When the TTL drops to zero, the execution is terminated with a special result code.

When the NIC partially offloads the egress path, the driver has to run the executor for any prefix of the table graph, such that the remaining suffix is offloaded. The driver can either precisely choose the TTL to make the executor stop at the right time, or just run the execution step-by-step.

The TTL field also protects from being caught in an infinite loop when the subsystem is misconfigured.