École Polytechnique Fédérale de Lausanne

Master Thesis

Advancing the State of Network Switch ASIC Offloading in the Linux Kernel

---

Student:
Andy Roulin (216690)

Direct Supervisors:
Roopa Prabhu, Director of Engineering for Linux Software
Academic Supervisors:
Prof. Edouard Bugnion
Prof Katerina Argyraki

March 2018

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Acknowledgements

I would like to thank first and foremost professor Ed. Bugnion for his amazing help throughout this thesis and my studies in general. I would not have made it through the tough times without remembering that you believed in me. Special thanks secondly to Roopa for being a great and patient teacher as well as friend. I could not have wished for a better supervisor. I would like to thank next my parents for being the awesome and lovely parents they are and putting up with me leaving my home country for a year to live the Silicon Valley experience. Thank you Emily for always being there for me and supporting me through this thesis. You know better than me now that "there is always a way". My grandfather, André, deserves special thanks for reassuring me through the hard times despite the fact that his grandson decided to live far away from home. You tried hard to convince me to stay. Special thanks to my friends and classmates Florian, Philémon, Jérémy and Eliéva, we got through these studies together. Finally, I would like to thank EPFL and the numerous amazing professors I met there who truly drove my need to understand how computers and software systems work, among others: Prof. Ienne, Prof. Beuchat and Prof. Candea.

Abstract

Modern data-center network operating systems rely on proprietary user-space daemons wrapping SDKs from switch vendors. Linux-based variants of these operating systems have benefited from increasing and simplified dataplane offloading support in recent years: kernel resources such as routes and next hops are offloaded to hardware and the kernel can, e.g., learn new MAC entries seen from the hardware forwarding plane. However, the Linux kernel in these operating systems has to be extended and customized as it still lacks constructs (constructs exposed to user-space or in-kernel constructs) needed for complete control plane processing and dataplane offloading. Managing limited hardware resources and keeping the kernel and hardware forwarding planes equivalent are two examples of challenges where the lack of appropriate constructs forces switch operating systems to use user-space solutions coupled with proprietary SDKs and custom kernel modifications.

Filling the gaps, i.e., designing and adding the missing constructs, would enable faster control plane processing (less user-kernel context switches) and faster dataplane configuration. It would also encourage in-tree kernel drivers from hardware vendors instead of the current closed-source user-space SDKs which would also improve performance while establishing Linux as the standardized API for network switches.

This thesis explores the different designs, performance challenges and trade-offs of completing the in-kernel switching API, starting from switch port configuration, faster control path packet processing and ending with in-kernel ASIC resource management. As most hardware vendors are not yet ready to open their drivers in the kernel, workarounds to still provide support for vendors' SDKs through the same in-kernel API are presented as well. A user-space switch port driver for Mellanox switches was ported to kernel space in order to accelerate control plane processing and bootstrap the in-kernel switch port configuration design. A prefetching scheme which hides PCI latency was designed to manage limited and shared ASIC resources with synchronous feedback to user-space daemons in case of exhausted resources.

Contents

List of Figures

1 Introduction

1.1 Linux in the Data Center

The widespread adoption of Linux in the server ecosystem is one of the most important success of the open-source operating system [14] [15]. It fueled the data center revolution from locked scale-up and monolithic servers to agile scale-out and open ones. Linux is the operating system of choice for modern data-centers servers where the choice of applications, hardware and Linux distributions is left to the data-center architects and administrators. Such a revolution has not happened yet in the networking part of data-centers. Network switches interconnecting the servers run proprietary operating systems on top of locked platforms. Disaggregation of data-center equipment is just starting and being pushed by companies like Cumulus Networks who offers to the industry an open-source Debian-based network operating system for switches, with hardware-accelerated IPv4/IPv6 routing and switching, which can be installed on industry-standard hardware.

1.2 Linux for Network Switches

Choosing Linux as the network operating system running on routers and switches makes sense: it already has built-in switching constructs the kernel, e.g., bridging and bonding drivers; it can forward IPv4 [27] and IPv6 [28] packets natively with the kernel's routing tables (using the sysctl option `net.ipv{4,6}.ip_forward`) and many routing suites such as Quagga [25] or the newer FRRouting [26] implement and manage IPv4 and IPv6 routing protocols. Furthermore, network administrators can use the well-known Linux command line syntax as well as its standard networking tools to configure their switches.

However, Linux only supports software-based switching and routing: forwarding of packets is done in software in the kernel. Data-center networking require much higher forwarding bandwidth than what software forwarding can achieve. The consequence is that Linux needs support for switches that implement packet forwarding in hardware with custom ASICs. Although only a few switch manufacturers (e.g., Mellanox [13]) have in-kernel Linux drivers, most vendors provide closed-source user-space SDKs to program their ASICs on Linux. This is only a part of the solution though: users and routing daemons interact with the kernel API to configure networking, not the SDKs. Most Linux-based network operating systems (NOS) work around that issue by building their own set of networking tools and their own networking stacks bypassing the kernel and interacting directly with the vendors' SDKs. This is not the approach taken by Cumulus Linux [29], the operating system by Cumulus Networks, which wants the Linux kernel to be the standard NOS for network switches. Users should interact with the kernel using the standard networking API and the kernel should program the ASICs accordingly. Therefore the Linux kernel needs switch ASIC offloading support to link kernel networking updates and ASIC control/data plane programming.

In this thesis, we review the current state of switch ASIC offloading in the Linux kernel and present new constructs to be used, e.g., a new switch port configuration scheme, and challenges to be taken care of, such as managing limited ASIC resources, in order to complete the in-kernel switch ASIC support. For networking areas that we only lightly cover, we provide insights for the future APIs and changes needed.

1.3 Thesis Outline

The thesis starts with a background section on network switches reviewing their architecture and their various purposes (section 2). It also describes a generic switch ASIC pipeline and the type of hardware resources used in today's high-end network switches. Section 3 describes the APIs and tools used today on Linux for switching and routing. Section 4 describes the current architecture used by Cumulus Linux to support switch ASICs and the future Cumulus Linux *ideal* architecture. It also describes intermediary steps taken to provide more in-kernel offloading support while still supporting closed-source vendors' SDKs. Control path packet processing is discussed in section 5 where an in-kernel network driver for Mellanox switches is presented (written during this thesis) to accelerate control packets handling. In-kernel and standardized switch port configuration is presented in section 6. Section 7 tackles the challenge of managing limited and shared hardware resources with synchronous feedback to the user. Related work is presented in section 8 while section 9 concludes.

1.4 Contributions

The different research projects, designs and implementations described in this thesis were done at Cumulus Networks, a Linux software company headquartered in Mountain View, California, as part of a Master's thesis internship. Cumulus Networks develops a Linux-based network operating system for data-centers and enterprise network switches. We list here the different contributions done during this thesis. Some of these contributions will be incorporated in the next release of the operating system while others are still work in progress or their designs might evolve in the future.

The current Cumulus Linux architecture works around limitations in the kernel with user-space SDKs and a proprietary user-space switching daemon in order to construct a complete network operating system. In section 4, we describe a new design for Cumulus Linux which moves switch port configuration, control path processing and ASIC dataplane offloading to the kernel. The end goal is for the Linux kernel to be a complete network operating system on its own, without requiring additional custom modifications and kernel bypasses. This design is work in progress for future versions of Cumulus Linux.

A network driver for Mellanox switches was ported to Cumulus Linux, based on a GPLv2 driver provided by Mellanox for their own custom OS. This port is presented in section 5. This port accelerates control flow processing by reducing user-kernel context switches and eliminating the use of a switching daemon in the control path. This driver will be included in the next release of Cumulus Linux.

In section 6, we present a new in-kernel switch port configuration scheme. Previously, configuring switch ports was done from the user-space SDKs and switching daemon. The new scheme detects and configure switch ports directly at boot-time using platform information and can be further configured using standard Linux networking tools. The Mellanox network driver, presented in the previous section, was adapted to this new configuration scheme.

Finally, section 7 tackles the resource management challenge. Switch ASICs have limited hardware resources which are shared between the different network resources (e.g., routes, MAC entries). The content of this chapter was published in a paper and presented at the Netdev 2.2 conference (Technical Conference on Linux Networking) in Seoul, South Korea, in November 2017 [24]. The section explores optimized dynamic partitioning of resources with PCIe latency-hiding

prefetching schemes. Synchronous feedback to user-space daemons is one of the core problems of the paper as well: routing daemons have to be informed synchronously if new routes cannot be offloaded to hardware because of exhausted hardware resources (which is not the case today).

2 Network Switch Architecture and Functionality

Network switches are at the core of Ethernet networks, the Internet or intranets. They enable computers to communicate in home/office local area networks (LANs), route packets in the core Internet's backbones and interconnect servers in high-end data-centers. In a home setup, the speed of your network is typically not limited by the performance of the router or switch. Thus, these home switches usually implement their functions in completely in software. For high-end data-centers where the number of interconnected servers grows faster and faster (scale-out design), forwarding packets has to be as fast as possible. Therefore, high-end switches implement their functionality in hardware with a switch ASIC forwarding packets close to line-rate. This section gives the reader background information on network switches and their ASICs for high-end performance scenarios. The section starts by describing the different functions implemented by a switch and then reviews the architecture of a switch with a top down perspective, from the motherboard architecture to the internal hardware tables used in switch ASICs.

2.1 What is a Network Switch?

A network switch (or Ethernet switch) is a networking device which links together Ethernet [30] devices, i.e., IEEE 802.1 or 802.3 devices, by relaying Ethernet frames between the connected devices [1]. It transports frames from a logical input channel, originating from one of the switch physical port, to the logical outgoing channel, ending with another switch physical port which corresponds to the packet's destination. The switching decision, i.e., deciding which port the packet should be forwarded to, was initially only based on L2 MAC addresses to relay Ethernet frames. Modern switches also perform routing based on L3 addresses (IP addresses) or even take multiple layers into account (TCP ports or flow numbers as an example) [2].

Figure 1 shows an example of a modern switch. This particular high-end switch targets spine and top-of-rack (ToR) use cases to connect high-throughput and high-density servers in data-centers. It comes with 32 100GbE ports and is said to carry a throughput of 6.4Tb/s and 9.52Bpps processing capacity [3].

Figure 1: Mellanox SN2700 32-port 100GbE Open Ethernet Switch [3]

2.2 Networking Functions Performed by a Switch

2.2.1 Bridging

Historically, network switches only performed Ethernet switching, also known as bridging, defined in the IEEE 802.1d standard [31]. Before switches, Ethernet hubs performed primitive Ethernet frames relaying by copying the received frame on all output ports, assuming to reach the destination on one of them.

Bridges, on the other hand, look at the destination MAC address and forward the frame to the output port on which the destination device is connected. The mapping table between ports and destination MAC address, called the Forwarding database (FDB), is constructed and updated on the fly by address learning. The bridge learns which devices are on which ports by looking at the source MAC address of packets it forwards. If no output port is registered for a destination MAC address, the bridge forwards the frame on all output ports (a process called "flooding"). Figure 2 depicts an example of a network switch used to bridge multiple computers while Figure 3 shows some possible state of the switch FDB table.

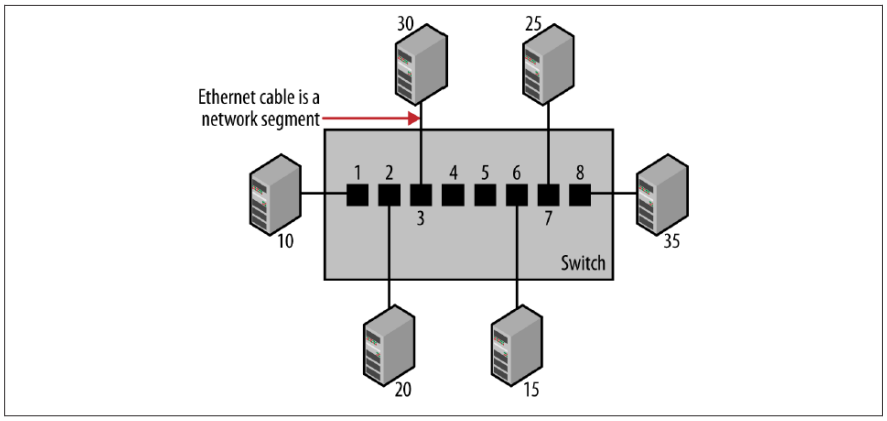

Figure 2: Network switch acting as a bridge [1]

Port	Station
1	10
2	20
3	30
4	No station
5	No station
6	15
7	25
8	35

Figure 3: Example Forwarding Database (FDB) maintained by the switch [1]

The IEEE 802.1q standard [32] defines Virtual LANs (VLANs) which extend bridging with the ability to form groups in a LAN and keep members of the different groups (i.e., VLANs) separate from the other. This is achieved using a VLAN tag, which includes a VLAN ID, inserted in the L2 frame header.

2.2.2 Routing

A modern network switch can perform switching at the Layer 3 of the OSI network model, also called routing. When acting as a router, the switch unpacks the L2 Ethernet frame to look at the destination IP address. The destination IP address can either correspond to a host on the same subnet or on a different subnet (reachable via one or multiple next-hops). In the former case, the host route table will contain the exact destination IP together with the MAC address and output port where the host is located. In the latter case, a longest-match lookup in the router's LPM table will return the next-hop's information.

2.3 Network Switch Architecture

Figure 4 shows the motherboard's components of a network switch. While based on the Mellanox SN2700 [3], it is representative of the architecture of network switches in general. A switch ASIC, forwarding network packets in hardware, is controlled by a processor running a network operating system. The rest of the architecture consists of I/Os components, memory and and physical ports.

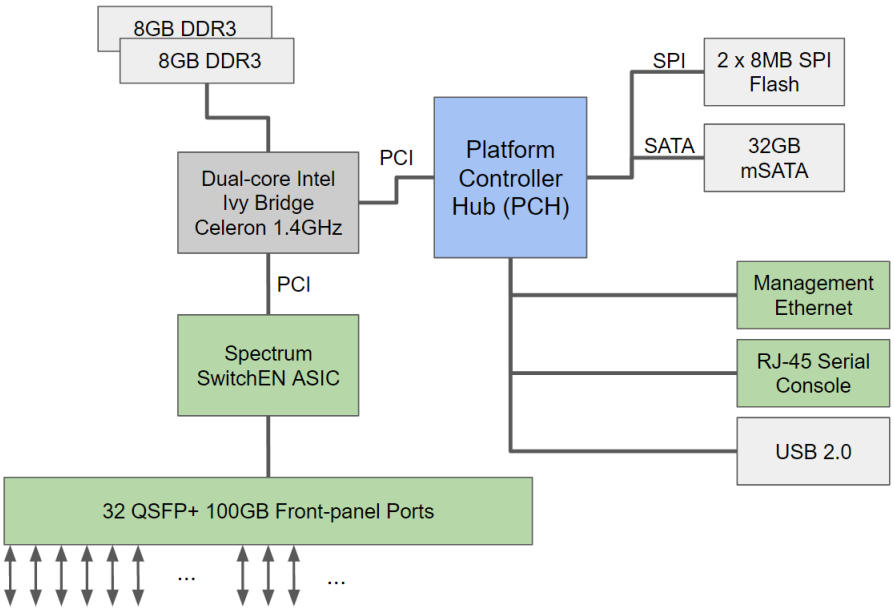

Figure 4: Network Switch Block Diagram

A more detailed description of the components involved is given below, following the path of a received control packet. A control packet has to be processed by the control plane protocols running on the CPU.

- Front-panel ports: usually QSFP+ ports, directly managed by the switching ASIC, which receives and transmits packets. The received control packet arrives one of the ports.

- Switch ASIC: the actual ASIC designed by the hardware vendor, e.g., Mellanox, which performs the data plane operations and forwarding. Non-control traffic is received from input ports and is directly forwarded to an output port. Control packets are passed to the CPU.
- Central Processing Unit (CPU): commonly x86/amd64 or ARM processors. It runs the switch operating system which handles control and management planes' operations; The switch ASIC driver receives the control packet and passes it to a user-space protocol daemon which processes the control information and updates its state. It might for example update the kernel routing tables.

The rest of the components accompanies the CPU in its operations and lets the switch administrator configure the switch.

- Main Memory: 16GB DDR3 is common for high-end switches;
- I/O: Solid-state drive (SSD) or hard disk drives (HDD) for storage; USB 2.0; Flash memory
- Management Ethernet: out-of-band Ethernet connection (usually named eth0 on the switch) for ssh connections and configuration of the switch through command line;
- RJ-45 Serial Console: additional way to connect and configure the switch through, e.g., a telnet connection;

2.4 The Three Planes of a Network Switch

All functions of a network switch can be divided in three different *planes*, as shown on Figure 5.

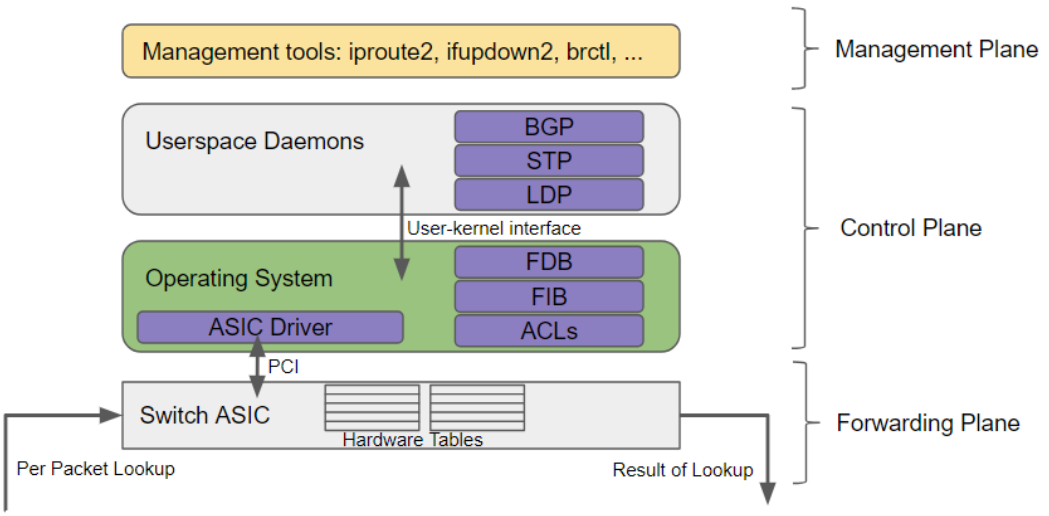

Figure 5: Forwarding, Control and Management Planes

2.4.1 Management Plane

The management plane consists of all the functions used to configure and monitor the switch device. Through the management Ethernet port of a switch (see subsection 2.3), commonly named eth0, an operator can ssh to the switch and for example:

- bring interface ports up or down;
- bridge ports together and configure VLANs;
- insert static FDB or route entries; and/or
- set up ACLs to filter traffic.

2.4.2 Control Plane

The control plane determines how packets are forwarded, i.e., which path through the switch to use. It consists of network protocol daemons (routing protocols: BGP, OSPF; STP for bridges; LDP; etc.) processing control packets coming from other switches to update the forwarding plane configuration. Control packets are protocol packets, e.g., STP packets, which are destined to the protocol daemons to help them make the most efficient forwarding decisions. Thus, when a switch receives a control packet on an input port, the packet will be forwarded to CPU for further processing by the operating system and the corresponding protocol daemon. Protocol daemons take into account the new information provided and update the switch ASIC data plane configuration accordingly.

2.4.3 Data Plane (or Forwarding Plane)

The data plane carries the actual traffic, i.e., forwards network packets reaching the switch. In contrast to the control plane, the CPU is not involved in the data plane. The data plane is being configured and updated regularly by the control plane but as the forwarding decisions to make are encoded in the ASIC's hardware tables, the ASIC can forward packets itself directly in hardware at line-rate.

2.5 Switch ASIC Architecture

The switch ASIC performs the data plane forwarding at line-rate, i.e., switches packets from the input port to the output port based on the data plane configuration given by the operating system and the protocol daemons. We describe in this section the generic pipeline stages that are used in switching ASICs. Although the implementation details are kept secret by hardware vendors, some information is available through white-papers [4] [5]. We can also claim that as time goes on, all switch ASIC implementations will be similarly implemented as the end goal functionality is the same and performance is the number one requirement.

A switching ASIC is organized around a switching fabric which interconnects the different physical ports together and can forward a packet between any pair of these ports. We will assume here that the interconnection network is a crossbar network although many other types of interconnections can be used and influence cost, scalability and performance trade-offs, e.g., Clos network, shared memory, bus-based.

Before delving into more details for each pipeline stage, we briefly summarize the various steps a packet will go through as follows: a packet is received on an input port and first goes through **(1) ingress processing**. The goal of the ingress processing stage is to decide to which output port the packet should be forwarded to. The packet is then scheduled for transmission through the **(2) crossbar network**. Whenever the desired channel is available, the packet is transmitted and enters the **(3) egress processing** stage connected to the final output port. After egress processing, the packet leaves the switch through the chosen output port (or it can leave on multiple output ports in case of multicast transmission).

Packet buffering/queuing can occur both at the ingress and egress stages, depending on the implementation and the desired trade-offs choices (head-of-line blocking can occur for ingress buffering while egress buffering is more difficult to implement [2]).

2.5.1 System-on-Chip Switch ASIC

Switch ASICs' silicon implementations typically rely on the concept of a **slice**: a micro-architectural component containing the ingress and egress packet processing silicon for a small number of ports. These slices are repeated throughout the ASIC to cover all the physical ports and interconnected through the global interconnection network, e.g., a crossbar network. This is shown on Figure 6. This figure shows a single system-on-chip (SoC) switch ASIC, i.e., the whole switching ASIC is implemented on chip on the device.

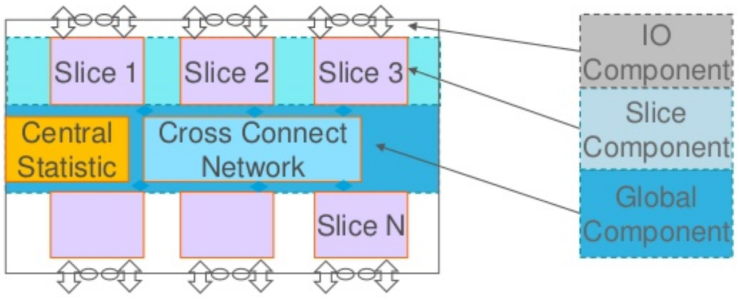

Figure 6: Switch ASIC single System-on-Chip Design [4]

2.5.2 Distributed Switch ASIC

More advanced switch ASICs distribute the slices across the motherboard. On Figure 7, from an Arista 7500 switch, the slices, called "Packet Processors" are actually implemented each on different chips and the interconnection network runs across the switch motherboard.

Going even further, this distributed architecture can be spread across multiple rack units while still interconnecting all slices. Multiple racks, like the ones shown previously on Figure 7, are interconnected across motherboards to end up with a similar design to the one shown on Figure 8. In this design, all slices are considered to be part of the same switch unit. For example, if one slice learns a new MAC address entry for its FDB table, the new learned entry will be known to all slices on the rack units composing this distributed switch.

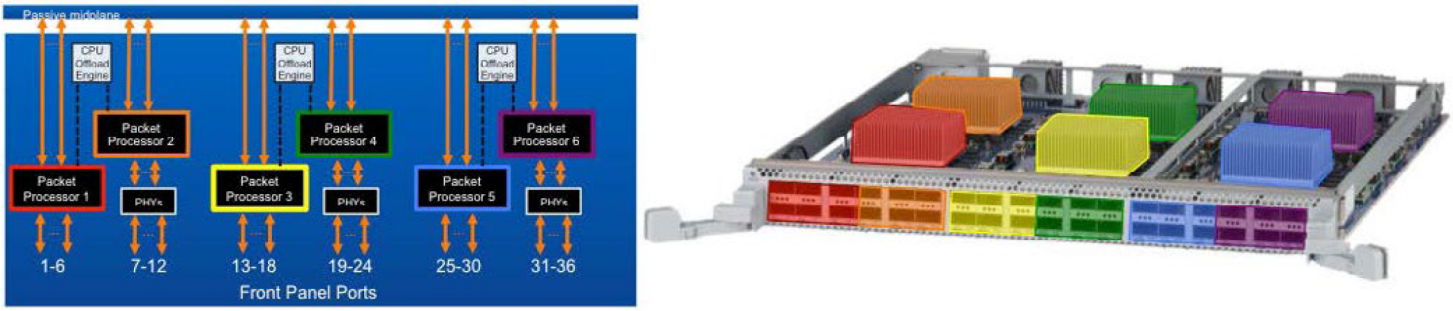

Figure 7: Switch ASIC Multiple System-on-Chip Design [5]

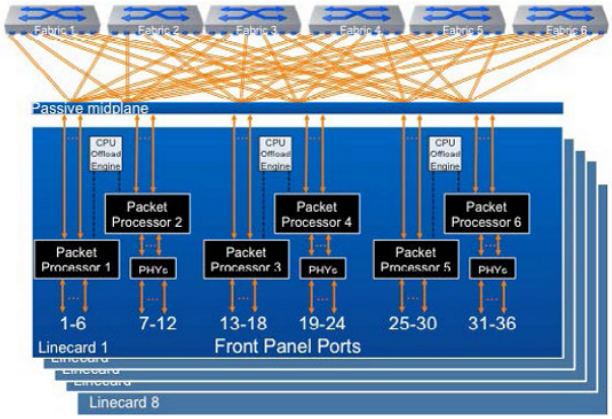

Figure 8: Distributed Switch ASIC Design (Multiple Rack Units) [5]

2.6 Switch ASIC Pipeline

The internal ASIC pipeline implementation is different for every switch ASIC on the market and is kept secret by the vendors. However, the more an operating system knows about this pipeline and the hardware resources involved, the more efficient optimizations and accurate resource management it can achieve. In the end, high-end network network switches target the same requirements and are subject to the same technical issues which means their designs typically follow a similar pipeline involving the same set of hardware resources and computation stages. We describe here a generic switch pipeline whose details and stages will vary and differ for a specific switch on the market.

As seen in the previous section (section 2.5 on switch ASIC architecture), the ASIC micro-architecture can be seen as a set of identical slices (containing ingress and egress processing) interconnected by a crossbar network. These slices can be broken down further into individual components (each slice contains the following four different components):

- Ingress Forwarding Controller: receives a packet and parses its header; makes the forwarding decision;
- Ingress Datapath Controller: schedules its transmission (and replication if needed) via the

crossbar network to the slice where the egress port is located;

- Egress Datapath Controller: receives the packet from the crossbar network; buffers the packet if needed; can replicate the packet in case it should be sent on multiple output ports, e.g., multicast transmission;
- Egress Forwarding Controller: takes care of sending the packet on the final output port

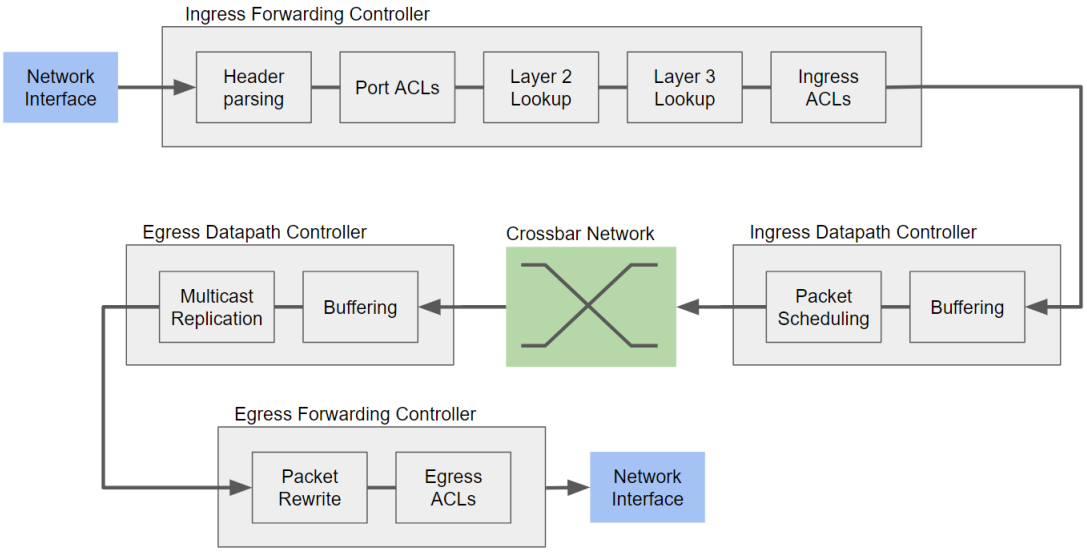

Figure 9: Switch ASIC Pipeline Stages

We now describe in more details every stage of the pipeline, from the ingress network interface to the egress output port, in order to understand exactly how a network switch decides how to forward a packet and what are the hardware resources (especially memories, i.e., tables) involved.

2.6.1 Ingress Network Interface

The ingress network interface stage implements the PHY and MAC layers of the OSI model. The PHY layer encodes bit streams from electrical pulses coming from the connector. The MAC layer constructs a frame from the bit streams and performs rudimentary error checking. The L2 frame constructed then enters the ASIC packet processing pipeline.

2.6.2 Ingress Forwarding Controller

The first micro-architectural component of a slice receives a packet from a network interface attached to it, parses the packet's headers and use these headers to make the forwarding decision, i.e., decide which output port the packet should be sent to.

Packet Header Parsing: Forwarding decision about a packet are done based on the packet's metadata which are contained in the headers. The header parsing stages extracts and saves the first 128 bytes of a packet which contain most importantly:

- L2 source and destination MAC addresses (SMAC and DMAC)
- VLAN ID
- L3 source and destination IP addresses (SIP and DIP)
- TCP headers

Port ACLs: A first set of filtering rules can be applied on a per-ingress-port basis to drop unwanted packets before they even entered the forwarding decisions stages. The ACLs are stored in an hardware ACL table, typically TCAM (Ternary Content-Addressable Memory).

L2 Lookup: The first decision a switch has to make is whether to switch (L2 forwarding) or route (L3 forwarding) the received packet. This decision depends on the DMAC. If the DMAC is equal to the switch's own MAC address then the packet has to be routed, otherwise the packet is to be switched.

In the latter case, the ASIC performs an **exact-match lookup** for the *(Destination MAC, VLAN ID)* tuple in its FDB hardware table. In case of a match, the hardware returns the output port whose network segment the final destination is located on. If there is no match, i.e., the switch does not know about the location of the device with the destination MAC address, then the packet is flooded on all ports.

MAC learning also happens in this stage: the ASIC learns the SMAC, maps it to the input port and fills the FDB table with this new entry.

L3 Lookup: If the packet is to be routed, the ASIC looks up the (VRF, Destination IP) tuple in two different hardware tables. The lookups can be done either in parallel or sequentially depending on the implementation:

- host route table: this table contains hosts directly connected to the same subnet (/32 prefixes for IPv4 or /128 prefixes for IPv6). The lookup in this table is an **exact-match lookup** on *(VRF, Destination IP)*. If a match is found, the table returns the output port and MAC address of the host.
- Longest-Prefix Match table (LPM table) for remote hosts: The **longest-prefix match** on *(VRF, Destination IP)* returns the output port and next-hop IP address. If the next-hop MAC address corresponding to the returned IP address is not in the hardware FDB table, the ASIC performs an ARP request to obtain that information.

Virtual Routing and Forwarding (VRF): virtualization applied to FIB routing tables. A networking device or operating system having support for VRFs means it can support multiple separate FIB tables. Network switch VRF's implementations can be as simple as having separate FIB tables in hardware (and thus a limited amount of VRF instances available). Support for VRFs in the Linux kernel was added by David Ahern in 2015 [10].

Ingress ACLs: Ingress ACLs filter packets after the forwarding decision has been made, filters being based on L2, L3 or L4 fields. The ACLs are again usually stored in fast TCAM

memories.

2.6.3 Ingress Datapath Controller

The ingress datapath controller takes care of transmitting the packet to the corresponding output slice where the final output port is located. In some implementations, it performs buffering but in all cases it has to schedule the packet for transmission to the crossbar network.

Buffering: Buffering of packets can happen at the input slice and/or at the output slice. Packet queuing at the input slice is simpler to implement and if the packet fails to reach the destination slice, the transmission can be tried again during the next arbitration round. The main drawback of input buffering is head-of-line blocking where a packet stuck in the buffer prevents the following packets to be transmitted.

Packet Scheduling: Within rounds of arbitration, packets at that stage compete for a free channel in the interconnection network between their input slice and the destination slice.

2.6.4 Egress Datapath Controller

The egress datapath controller receives the packet from the crossbar network and, in some ASIC implementations, buffers it here until it is ready to leave the switch on the output port located on this same slice. The output slice is also a good place to replicate the packet transmission for multicast transmission as described below.

Buffering: Output buffering avoids head-of-line blocking and does not require arbitration in the interconnection network. However, the latter's implementation is more complex: packets must be transmitted through the interconnect as soon as they arrive at the input slice and are processed. With N input slices, the switching transfer must be at least N times the input slice reception and processing speed, otherwise packets will get lost due to the lack of input buffering [2].

Multicast Replication: Multicast replication is usually performed at the output slice as it is often the case that related multicast output ports are located on the same output slice. Replication at the output slice avoids replicating the packet before transmitting it through the crossbar network while most of the replicated packets would be destined to the same output port. Any replication that needs to reach another slice will be transmitted again through the crossbar network.

2.6.5 Egress Forwarding Controller

The final pipeline component, the egress forwarding controller, finally sends the packet on the destination output port after rewriting some of its metadata and applying egress ACLs.

Packet Rewrite: Before the packet can leave the switch, some of its header might have to be rewritten. For example, if the packet was routed, the destination MAC must be updated to be the MAC address of the next-hop and the Time-to-Live (TTL) must be decremented.

Egress ACLs: Lastly, another set of ACLs filters can be applied to packets right before they would leave the switch. These ACLs can depend on combined parameters such as the output port and L2/L3/L4 headers.

2.6.6 Egress Network Interface

The egress network interface stage has to transmit the packet on the chosen physical port. It implements the MAC and PHY layers of the OSI model. The MAC layer decodes the L2 frame to a bit stream and the PHY layer encodes the bit stream to electrical pulses going to the connector, effectively leaving the switch.

2.7 Switch Hardware Tables

After describing the general pipeline stages in a switch ASIC, we will now look at the hardware tables, i.e., memories used by the different stages. Switch ASICs resources are often shared between different types of objects. For example, the host routes table and MAC address table might use the same underlying memory, typically implemented with some form of a hash table. The entries and keys that use the table are defined in blocks that can be resized and thus allow routes versus MAC capacity to be traded off against each other. Furthermore, different types of host route entries and LPM entries exist (e.g., IPv4 entries, IPv6/64 and IPv6/128 entries) which can also share the same memory and be traded off against each other. This dynamic partitioning of shared resources as well of the presence of multiple separate hardware tables (static partitioning) is an opportunity for clever resource management schemes done at the operating system level. This challenging problem is described in thorough details in section 7 about resource management.

We illustrate shared hardware tables with the Arista 7500E series of switches [5]. The first type of hardware memory used is a hash table for **exact-match lookups**. This table is used on the Arista 7500E switch for the FDB table (MAC address lookups), host routes tables for IPv4 and IPv6 as well as multicast routes lookups. This table is shown on Figure 10.

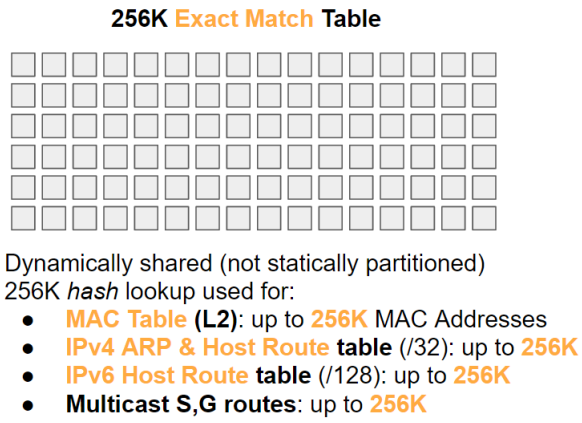

Figure 10: Exact-match Table in the Arista 7500E switch ASIC [5]

Longest-prefix match for IPv4 uses its own separate hash-based memory to lookup IPv4 routing prefixes, as shown in Figure 11. Some other ASIC implementations will use TCAM-based memories for IPv4 LPM lookups as these ternary-content memories are particularly well-suited for longest prefix matches lookups (see next paragraph for an explanation).

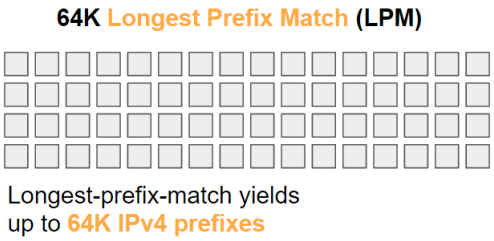

Figure 11: Longest-prefix Match Table in the Arista 7500E switch ASIC [5]

A shared high-speed TCAM memory is typically used for ACLs entries and IPv6 **longest-prefix match**. The ability of a TCAM to store three different inputs (0, 1 and X) allows for wild-card bit matching and thus matching on only parts of header fields and finding the longest match for a routing prefix are achieved easily (at the expense of a more costly memory) with TCAMs.

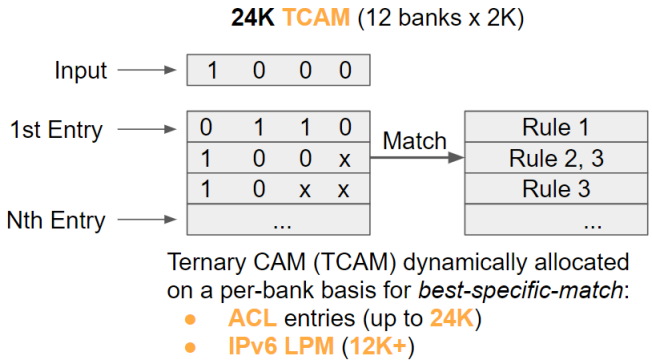

Figure 12: TCAM Table in the Arista 7500E switch ASIC [5]

3 Linux for Switches

3.1 Current State of Operating Systems for Switches

The current networking landscape is dominated (more than 50% of the market share [17] of switches and routers) by *Cisco Systems* and the different Cisco OSes available for switches: IOS, NX-OS. Competitors like Juniper, Arista, Dell and HP also provide their own network operating systems: some are based on a Linux kernel, some on FreeBSD and some operating systems are custom. For example, Cisco IOS is a monolithic in-house operating system while the new NX-OS [16] is based on a Linux kernel but keeps Cisco's commands API to configure switches.

3.2 Linux Constructs for Switching

The Linux kernel today can already function as a software switch and perform switching, routing as well as traffic filtering/shaping using individual NICs. We describe here the main user-space tools used for such network functions. Keeping the same interface and semantics while offloading the forwarding plane to the switch hardware is the main idea behind Cumulus Linux.

3.2.1 Switch Port Configuration

The switchdev model adopted by the Linux kernel to support network switches exposes switches' physical ports as netdevices (`struct net_device` in the kernel) for user configuration. Netdevices are also known as "network interfaces" and listed when using network commands such as `ip link show` or `ifconfig`.

```
root@mlx-2700-04:~$ ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group
    default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT
    group default qlen 1000
    link/ether 7c:fe:90:6d:b9:b8 brd ff:ff:ff:ff:ff:ff
3: swp1s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master br0 state UP
    mode DEFAULT group default qlen 1000
    link/ether 44:38:39:00:a5:bc brd ff:ff:ff:ff:ff:ff
4: swp1s1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master br0 state UP
    mode DEFAULT group default qlen 1000
    link/ether 44:38:39:00:a5:bd brd ff:ff:ff:ff:ff:ff
5: swp1s2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master br0 state UP
    mode DEFAULT group default qlen 1000
    link/ether 44:38:39:00:a5:be brd ff:ff:ff:ff:ff:ff
6: swp1s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master br0 state UP
    mode DEFAULT group default qlen 1000
    link/ether 44:38:39:00:a5:bf brd ff:ff:ff:ff:ff:ff
7: swp3: <BROADCAST,MULTICAST> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT group default
    qlen 1000
    link/ether 44:38:39:00:a5:b0 brd ff:ff:ff:ff:ff:ff
8: swp4: <BROADCAST,MULTICAST> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT group default
    qlen 1000
    link/ether 44:38:39:00:a5:b2 brd ff:ff:ff:ff:ff:ff
[...]
```

Listing 1: Output of `ip link show` on a Mellanox SN2700 switch running Cumulus Linux

Listing 1 shows a sample of the network interfaces on a Mellanox switch running Cumulus Linux:

- `eth0` is the management interface from which the ssh connection is established;
- the first physical port (`swp1`) has been split into four sub-interfaces (`swp1s{0,1,2,3}`). Mellanox switches support splitting physical ports into two sub-ports or four sub-ports. Splitting into four sub-ports uses the next physical port as well (`swp2` is not available).
- `swp1` sub-interfaces are all part of a bridge named `br0`

The user can query more information about an interface, set the speed and duplex mode (among others) of an interface using `ethtool`. Listing 2 shows some information that can be retrieved using ethtool. The `-i` option shows the netdevice driver used, here `sx_netdev` for Mellanox switches.

```
root@mlx-2700-04:~$ ethtool -i swp1s0
driver: sx_netdev
version: 1.0
[...]
root@mlx-2700-04:~$ ethtool swp1s0
Settings for swp1s0:
        Supported ports: [ TP FIBRE ]
        Supported link modes:   1000baseT/Full
                                10000baseT/Full
                                10000baseCR/Full
                                10000baseSR/Full
                                10000baseLR/Full
                                10000baseLRM/Full
                                10000baseER/Full
        Supported pause frame use: Symmetric
        Supports auto-negotiation: Yes
        Supported FEC modes: None BaseR RS
        Advertised link modes:  10000baseT/Full
        Advertised pause frame use: Symmetric
        Advertised auto-negotiation: Yes
        Advertised FEC modes: Not reported
        Speed: 10000Mb/s
        Duplex: Full
        Port: Other
        PHYAD: 0
        Transceiver: internal
        Auto-negotiation: off
        Link detected: yes
```

Listing 2: Information retrieved by `ethtool` on switch port netdevices

3.2.2 Bridging and Bonding

Users can bridge and bond network interfaces, thus creating virtual interfaces. The configuration depends on the particular Linux distribution and Cumulus Linux being based on Debian, the network configuration is to be found in the `/etc/network/interfaces` file. Listing 3 shows a sample extract of this file with a bond and a bridge as well as the output of the `brctl` command.

```
root@mlx-2700-04:~$ tail -12 /etc/network/interfaces
auto bond0
iface bond0
    bond-slaves swp5 swp6

auto br0
iface br0
    address 11.0.0.10/24
    bridge_ports swp1s0 swp1s1 swp1s2 swp1s3
    bridge_stp on
    bridge_maxwait 20
    bridge_ageing 200
    bridge_fd 30

root@mlx-2700-04:~$ brctl show
bridge name     bridge id               STP enabled     interfaces
br0             8000.44383900a5bc       yes             swp1s0
                                                        swp1s1
                                                        swp1s2
                                                        swp1s3
```

Listing 3: Bonding and Bridging on Debian/Cumulus Linux

3.2.3 Routing

The Linux kernel maintains FIB tables for IPv4 and IPv6 with next-hop and output port information for routing prefixes. These tables are mostly populated by routing daemons (e.g., Quagga, FRR) using routing protocols such as BGP or OSPF to exchange prefixes and routing topology between routers. Users can add routes manually as well using the iproute2 syntax `ip route add`. Listing 4 shows an example kernel routing table where entries are marked as `offload` when offloaded to a underlying switch device. The reader will note that most routes were added by the Zebra routing daemon and that the 12.0.0.2 prefix has two next-hops available of equal weight, using Equal-cost multi-path routing (ECMP).

```
root@mlx-2700-04:~$ ip route show
    default via 192.168.0.2 dev eth0
    11.0.0.0/30 dev sw1p1  proto kernel  scope link  src 11.0.0.2 offload
    11.0.0.4/30 via 11.0.0.1 dev sw1p1  proto zebra  metric 20 offload
    11.0.0.8/30 dev sw1p2  proto kernel  scope link  src 11.0.0.10 offload
    11.0.0.12/30 via 11.0.0.9 dev sw1p2  proto zebra  metric 20 offload
    12.0.0.2  proto zebra  metric 30 offload
        nexthop via 11.0.0.1  dev sw1p1 weight 1
        nexthop via 11.0.0.9  dev sw1p2 weight 1
    12.0.0.3 via 11.0.0.1 dev sw1p1  proto zebra  metric 20 offload
    12.0.0.4 via 11.0.0.9 dev sw1p2  proto zebra  metric 20 offload
    192.168.0.0/24 dev eth0  proto kernel  scope link  src 192.168.0.15
```

Listing 4: Routing Table on Cumulus Linux

3.3 Offloading Network Switch Functions

Offloading the functions of a network switch (or of a network device in general) requires two components:

- **Network Driver:** A network driver exports each switch port as a `net_device` as if every port was a single NIC device on its own. The network driver handles interrupts and transmission/reception of frames on individual ports. For switches in the Linux kernel, the network driver is split between the `net_device` driver, which handles the netdevice

configuration and operations, and the ASIC driver, which takes care of interrupts, packet reception/transmission and interacting with the SDK/netdevice driver.

- **Support for Offloading Operations:** This support is required to program the ASIC forwarding plane corresponding to the kernel forwarding plane. Without it, the forwarding would happen in software, e.g., bridging would happen in the Linux bridge driver and all forwarding traffic would be required to come to CPU. In order to offload operations, the kernel needs to notify the hardware of new dataplane configuration changes. Therefore hooks and/or new device operations need to be added. When a user adds a new MAC entry to the FDB, the ASIC driver needs to be notified in order to offload this entry to hardware. Conversely, if a new MAC entry is seen from the traffic in hardware, the kernel needs to be notified as well in order to add this entry to the kernel FDB and keep kernel/hardware state in sync.

The network driver part is discussed in details in section 4 and a netdevice driver implementation for Mellanox switches is presented in section 5. The support for offloading switch operations has been started with the switchdev initiative, described below and is still under active discussion by members of the switch community.

3.4 The Switchdev Initiative

The switchdev initiative [7] was started in 2014 by Cumulus Networks. It aims at designing an in-kernel driver model for Ethernet switches and defining the hooks and APIs needed to support offloading operations.

Switchdev started with hooks through the networking stack to offload networking objects (routes/FDB entries/etc.) to hardware with the main idea of exposing switch ports as Linux network interfaces to the user. The initial implementation assumed all networking objects would lead to a switch network device and from there to a switch port driver for hardware offload. The switch network device would represent the switch device itself, link the switch ports network interfaces together and thus provide an anchor point for offload. This global switch device was abandoned but the switch ports devices themselves still remain.

The switchdev API contains generic recursive `net_device` stack traversal helpers to reach the switch port themselves as they can be hidden behind other virtual devices, e.g., bridges or bonds.

There are currently two drivers in the kernel implementing the switchdev model:

- the Mellanox `mlxsw` driver for Mellanox Spectrum ASICs; and
- a driver for a QEMU virtual switch named Rocker which was used as the switchdev prototyping vehicle when initially designing the switchdev API [8].

The switchdev driver model is still under active development by Mellanox. However, being the only vendor having a switchdev driver in the Linux kernel, Mellanox drives the API design in directions that do not always correspond to what other vendors would need from such an API. Thus, switchdev was used as a starting point for the work in this thesis and in general for Cumulus Linux' next steps to support switches in the kernel.

3.4.1 Switchdev Offload Notifiers

Switchdev relies on the kernel notification subsystem to inform switch port drivers that a networking change requiring updating the hardware state has happened. For example, the switch port driver is notified when a switch port joins/leaves a bridge or a new route has been added to the kernel FIB table. This notification system is close to the switchd daemon model where switchd listens to Netlink notifications in order to update the underlying hardware. One major drawback of the asynchronous kernel notifiers is that it cannot make the notification fail and return an error to the caller. For example, if the hardware FIB table is full and a new route is added to the kernel, the routing daemon will not be notified of the failure. This problem is described thoroughly in section 7 about resource management.

4 Cumulus Linux Architecture: Present & Future

Cumulus Linux is the network operating system developed by Cumulus Networks. Using the Linux kernel with custom (openly available) modifications coupled with vendors' proprietary SDKs, it supports the most important switches on the market from, e.g., Broadcom (the main vendor on the market) or Mellanox. This section reviews its original architecture based around a switching daemon wrapping the hardware vendors' SDKs. In order to establish Linux as the standard platform for network switches (in a similar way that Linux was established as *the* computing platform for servers in the 90s), Cumulus Networks wants to push native support for switches in the kernel rather than relying on additional kernel extensions, user-space switching daemons and proprietary SDKs. The section continues with describing this ideal targeted architecture while also setting important milestones (or steps) to reach that goal.

4.1 Starting Point: tuntap-based ports with user-space SDK and switch daemon

Figure 13 shows the original Cumulus Linux architecture to support a switch. A Network card is also shown in the picture for comparison purposes. In the figure, the drivers are shows as separate from the kernel for emphasis purposes: the drivers might be out-of-tree drivers and/or proprietary. Nevertheless, as they are at least loaded as kernel modules and run in kernel-mode, they are actually part of the kernel.

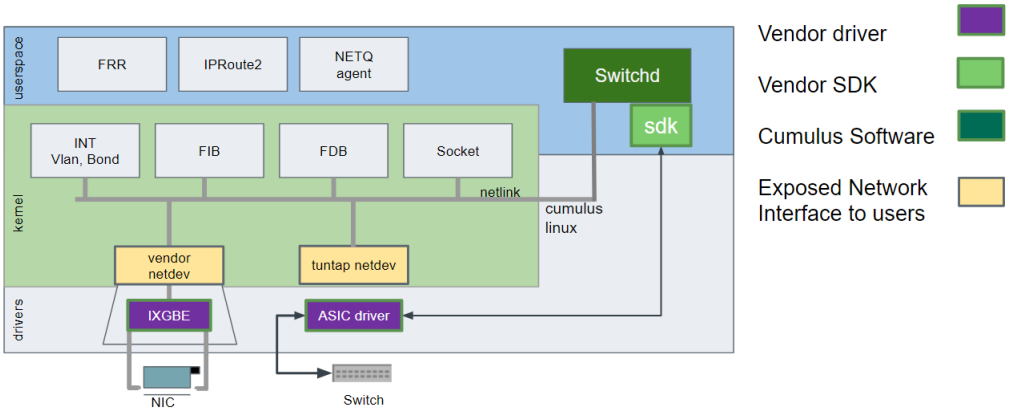

Figure 13: Original TUN-based switch infrastructure

As described in section 3.4, switch ports are to be exposed as `net_devices`, or Linux network interfaces to the user. Users can bridge a set of these interfaces together to create a Linux bridge and add VLAN IDs to these bridges. Similarly, routing daemons add routes to the kernel using standard Netlink user-space kernel channels. These new bridge constructs and routes needs then to be offloaded to the switch ASIC. However the Linux kernel does not have any data-plane programming API. Furthermore, the hardware vendors only provide user-space SDK to program their switching ASICs: port configuration and forwarding plane programming has to be done from user-space but users and routing daemons are talking directly to the kernel. Finally, hardware vendors do not usually provide a `net_device` driver for their ASICs, also called a

network driver, which expose switch ports as `net_devices`. The vendors only provide an ASIC driver which is able to receive/transmit packets to the ASIC and program the ASIC's forwarding plane. This ASIC driver only communicates directly with the SDK (through, e.g., `ioctls`) and do not involve network interfaces

To solve these multiple issues arising from a mismatch between the Linux kernel capabilities and network switches requirements, Cumulus Linux brings:

- a TUN interface (i.e., `net_device`) for every switch port
- a user-space switch daemon (called `switchd`) which listens to Netlink notifications from the kernel and calls into the vendor's SDK to program the ASIC accordingly.

4.1.1 TUN/TAP Interfaces

As vendors do not usually provide network drivers for their ASICs, Cumulus Linux creates one tun interface per physical port. Tun interfaces, from the TUN/TAP kernel extensions package [6], are virtual network interfaces which are backed by a user-space program, switchd in this particular use, instead of a real network adapter. The switch ports appear as `swp1, swp2`, ..., `swpN`, `N` being the number of ports.

TUN/TAP provides packet reception and transmission for user space programs. It can be seen as a simple Point-to-Point or Ethernet device, which, instead of receiving packets from physical media, receives them from user space program and instead of sending packets via physical media writes them to the user space program.
In order to use the driver a program has to open /dev/net/tun and issue a corresponding ioctl() to register a network device with the kernel. A network device will appear as tunXX or tapXX, depending on the options chosen. When the program closes the file descriptor, the network device and all corresponding routes will disappear (extract from kernel documentation [6]).

4.1.2 User-space Switch Daemon

The user-space switch daemon, `switchd`, is a proprietary daemon, written by Cumulus Networks, wrapping the also proprietary vendors' SDKs and listening to Netlink notifications to configure the underlying ASIC. The Netlink notifications come from any kernel update made, e.g., by users configuring the TUN-netdevs, routing daemons updating the kernel FIB or FDB entries aging in the kernel.

`Switchd` is also in the path of packet reception/transmission for packets that are destined to the switch itself, i.e., control packets. This is described in the next subsection.

4.1.3 Reception and Transmission of Control Packets

In this model, control packets cross the user-space kernel boundaries twice which harms performance. We will describe the receive path in more details (the transmit path is the same path but reversed):

1. A packet arrives at a switch port; the ASIC pipeline notices the packet is destined to the switch itself and forwards the packet to CPU.

2. The ASIC driver reads the packet from the RX DMA ring and transmits it to the SDK (**first kernel→user context switch**)
3. the SDK passes the packet up to switchd which transmits it to the corresponding TUN interface. (**second user→kernel context switch**).
4. The packet is now back in the kernel and is finally processed by the network stack.
5. After processing, the packet can be delivered by the TUN interface to the corresponding control daemon, e.g., a BGP routing daemon for BGP control packets.

4.1.4 Performance and Reliability Shortcomings of this Model

Two main performance issues arise from this TUN-based model:

- control packets cross the user-kernel interface twice, causing two context switches in the control path;
- forwarding plane programming also suffers from user-kernel context switches: a new route inserted in the kernel notifies switchd which calls back to the kernel for the ASIC driver to actually program the ASIC with the new route.

Furthermore, the complexity of this model leaves room for improvement: switchd has to maintain a cache of kernel network configuration and maintaining the kernel, switchd cache and hardware tables perfectly in-sync is difficult and causes many hard bugs to track. It is clear that this model to support network switches was designed **around** the Linux kernel rather than in the kernel itself (which was how to solve the problem, given the lack of appropriate offload API in the kernel).

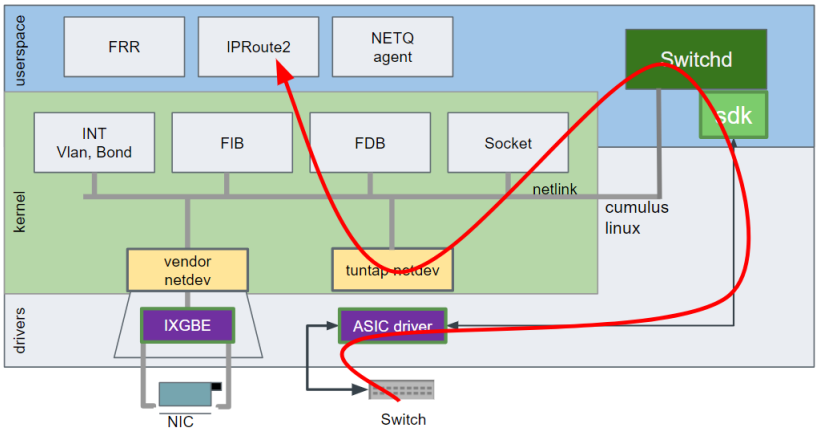

Figure 14: Path Taken by Control Packets

4.2 Ideal Goal: Kernel Switch Ports with Kernel SDK

To contrast with the current Cumulus Linux architecture, let us look at the ideal envisioned model, shown in Figure 15. In this model, the Linux kernel supports complete offloading of all networking functions required by a network switch:

- switch port configuration;
- control-plane packet processing; and
- data-plane programming

Netdevices corresponding to switch ports are created at boot-time using platform information stored in BIOS/UEFI ROM (similarly to common motherboard information in current desktop or server computers). The vendor network driver handles the netdevices and control packets. When a netdevice receives a control packet, it passes it directly to the network stack which delivers it to the user-space application without any additional user-kernel context switches. The switchdev model would add the missing data-plane programming APIs to offload data-plane forwarding from the kernel tables to the switch ASIC. The vendor switchdev driver would implement the new dataplane offload API for a particular ASIC. The user-space switching daemon and SDK are not involved anymore. The SDK functionality is moved and its functionality is spread among the vendors' ASIC, netdevice and switchdev drivers.

This model only shows the direction Cumulus Linux wants to steer its development. It is not realistic as it is unlikely that vendors would open their SDKs and also rewrite them as kernel modules. Furthermore, most vendors do not provide a netdevice driver. Thus two intermediary steps are envisioned for Cumulus Linux:

1. Adding netdevice drivers to process control path packets in the kernel instead of switchd and the SDK.
2. Supporting user-space SDKs for data-plane programming via the in-kernel data-plane offloading API (called switchdev in Figure 15).

4.3 Intermediary Step 1: Network port driver to accelerate control path processing

The first step is take control packet processing out of switchd, the switching daemon. In order to pass the packets directly to the kernel network stack, the processing done by the SDK in user-space must be moved to the netdevice driver. Thus the vendor must provide a netdevice driver, it cannot be the TUN/TAP connector anymore. The modified architecture with accelerated control path processing is shown on Figure 16. The next section, (section 5), describes the port of such a driver for Mellanox switches.

4.4 Intermediary Step 2: Trampoline driver to support user-space SDKs

The second intermediary step is a realistic step on the observation that it is unlikely that hardware vendors will move the ASIC SDK to a kernel module or even consider open-sourcing their drivers. Switch SDKs can be large software projects written in higher-level languages and the transition

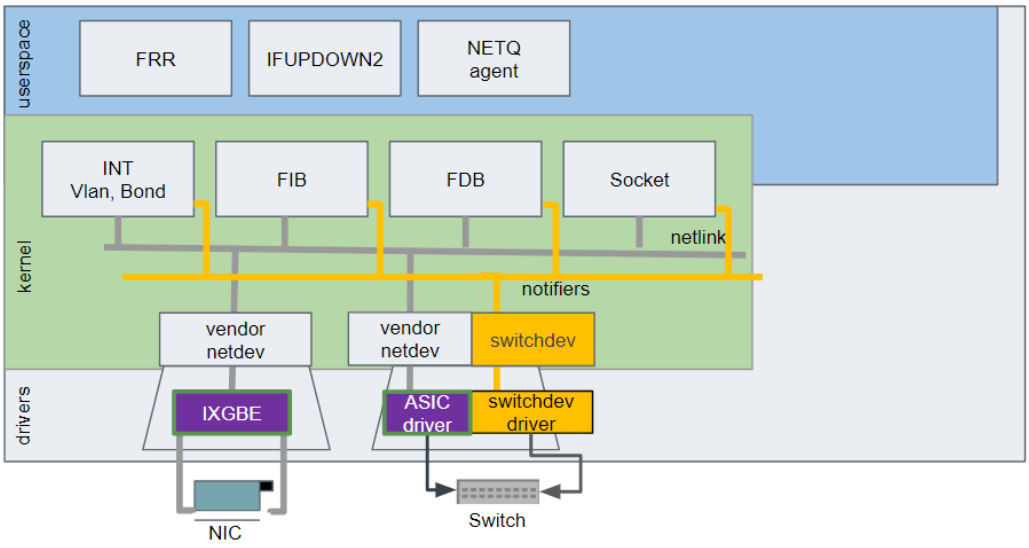

Figure 15: Ideal Cumulus Linux Architecture

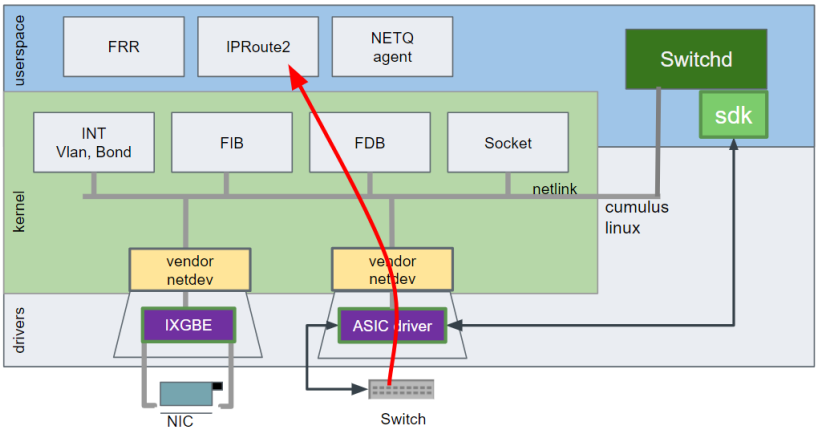

Figure 16: Modified Architecture with a Network Driver and Accelerated Control Path Processing

to a Linux kernel module can be costly or simply not attractive enough for switch manufacturers who already use another network operating system. Therefore, in order to still provide support for user-space SDKs while encouraging the use of a new in-kernel switchdev API for data-plane offloading, Cumulus Linux plans to add a trampoline driver which will forward switchdev calls to the SDK. Control plane is already handled by the netdevice driver from step 1 above.

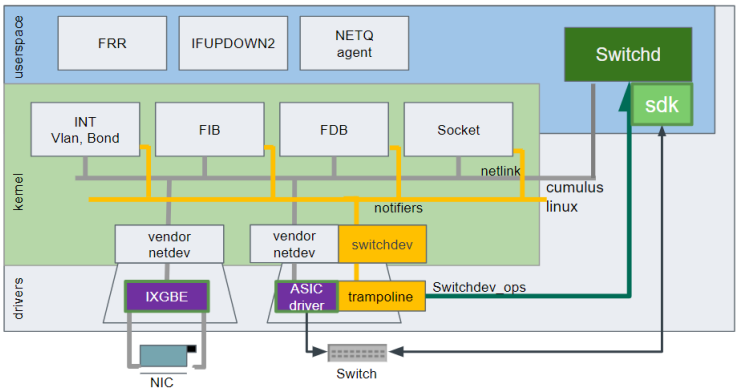

Figure 17: Trampoline Driver for Data Plane Programming via User-space SDK

5 A Mellanox Network Driver for Cumulus Linux

This section describes the port of a network driver, called `sx_netdev`, for Mellanox switches to the Cumulus Linux operating system. An equivalent port was done for Broadcom switches in 2015 while the Mellanox switches were still using the TUN/TAP-based netdevs until this new `sx_netdev` driver. This port adds kernel-based netdevices for control plane traffic through the GPLv2 `sx_netdev` network driver provided by Mellanox. This original driver is a netdevice driver (distributed as a Linux kernel module) but it does not interact with the Linux network stack. It was written for the Mellanox OS, which is based on Linux but bypasses most of the kernel networking code to use a custom user-space stack instead.

The `sx_netdev` kernel driver was thus modified to be used with the Linux networking stack for control plane packet processing (bypassing the swichd user-space switching daemon). Dataplane configuration is however still performed via switchd and the Mellanox SDK.

This section starts by presenting the TUN/TAP-based control plane packet flow and comparing it to the new kernel-based `sx_netdev` packet flow. It then lists the major advantages of the latter approach and presents the system's infrastructure changes needed to make this solution work, e.g., port netdevices creation has to be done from the kernel driver itself. The implementation details and issues encountered throughout the port are described next. Most of the issues arose from `sx_netdev` being originally written for the Mellanox proprietary OS whose implementation details and assumption on how it interacts with the `sx_netdev` driver are unknown. Ensuring parity with the TUN/TAP solution was also important to avoid any breakage for customers using Cumulus Linux. Finally the section ends with a performance review of this new model (4× RX and 2× TX TCP throughput increase).

5.1 Control Packet Flow

We start by reviewing the path that control packets take in both the old TUN/TAP model and the new kernel-based one.

5.1.1 TUN Packet Flow

Shown in Figure 18, the TUN packet flow for control path packets crosses the user-kernel space boundary twice. Crossing this boundary is expensive and it is not necessary as the packet really only needs to reach the control daemon it is related to. It is only due to the Mellanox ASIC driver's lack of interaction with the Linux Network Stack: the ASIC driver only knows how to transmit and receive packets from/to the SDK and does not use the concept of a netdevice which has to be used for the packet to go through the Linux network stack and be delivered to the final user-space packet processing application. The TUN/TAP devices are used as anchor points to represent each of the switch's physical port: users can reference them and configure them together as bridges, LAG interfaces or other networking constructs.

5.1.2 sx_netdev Packet Flow

With `sx_netdev`, the control path only crosses the user-kernel boundary once to reach the final control daemon. The control packet flow is shown on Figure 19 The packet is passed directly from

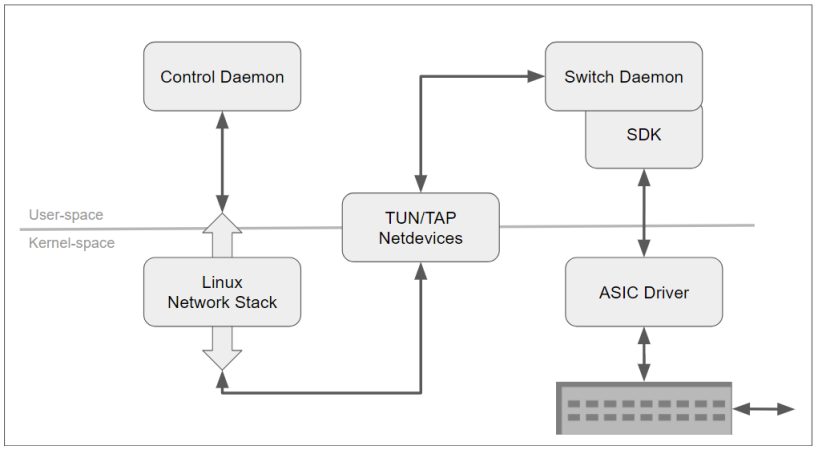

Figure 18: TUN-based Control Packet Flow

the `sx_netdev` driver, which understands and maintains netdevices for the switch, to the Linux networking stack. This improves performance with a cleaner design overall. Another advantage of this model is that it takes control packet processing out of switchd. Consequently, control packets would still flow in case switchd is brought down or has to be restarted and thus control daemons continue will continue to get control-plane updates.

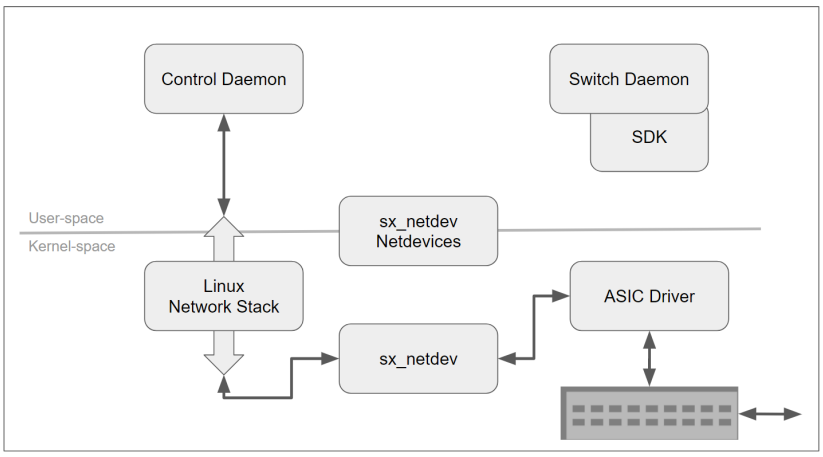

Figure 19: sx_netdev Control Packet Flow

5.2 Implementation

5.2.1 Port Netdevices Creation via Netlink

The TUN/TAP ports were created by the switchd daemon according to user-space configuration files describing the platform port topology, e.g., number and types of ports. With the new

sx_netdev model, the port creation is still done via switchd but it invokes Netlink's netdevice creation routines which are implemented by the sx_netdev kernel driver, shown in listing 5.

```
static struct rtnl_link_ops sx_netdev_link_ops __read_mostly = {
        .kind           = "mlx_sx_netdev",
        .maxtype        = IFLA_SX_NETDEV_MAX,
        .policy         = sx_netdev_policy,
        .priv_size      = sizeof(struct sx_net_priv),
        .setup          = sx_netdev_setup,
        .validate       = sx_netdev_validate,
        .newlink        = sx_netdev_newlink,
        .dellink        = sx_netdev_dellink,
        .get_size       = sx_netdev_get_size,
        .fill_info      = sx_netdev_fill_info
};
```

Listing 5: Route Netlink (rtnl) Operations Implemented in sx_netdev

Early at startup time, switchd uses the user-space port description files, still stored in the same location, to create a netdevice, through sx_netdev_newlink, for every physical port of the switch. For the next versions of Cumulus Linux, this platform information is planned to be moved to BIOS/UEFI storage with a standard for platform description common across all switches (see section 6 about platform description using APD).

5.2.2 Two-phase Switch Port Initialization

The sx_netdev_newlink implementation does very little work but creating and registering a barebone netdevice with the correct port name, e.g., swp1. The port name is passed as an argument by switchd. The rest of the netdevice and actual ASIC port initialization will be done by the SDK after switchd and the SDK will have initialized the ASIC itself. This two-phase initialization (netdevice creation and actual port configuration by the SDK), which is matched by an equivalent two-phase termination procedure, is similar to the TUN/TAP model where TUN/TAP netdevices are created ahead of time and persist during a switchd restart. This is important as Linux network constructs, such as bridges or bonds, reference netdevices and need to persist across switchd restarts. Thus a switchd restart does not delete and recreate the netdevices, it only does the second part of the port initialization again (ASIC port configuration).

5.2.3 Ethtool operations Support and Packet Statistics

The sx_netdev driver implements the ethtool operations needed for setting the speed/duplex/autonegotiation modes of network interfaces. It also implements getting statistics for packets received/dropped. These operations are not fully implemented in the sx_netdev driver itself as statistics retrieval and port configuration can only be done from the SDK. The operations actually call back to the SDK using a private channel between the kernel and the SDK/switchd. The call trace when a user invoke an ethtool operation is thus: ethtool → kernel → sx_netdev → switchd/SDK. Listing 6 shows the ethtool operations added to the sx_netdev driver. Functions in that listing starting with the prefix port_* are operations forwarded to the user-space SDK.

```
static const struct ethtool_ops sx_cl_ethtool_ops = {
        .get_link_ksettings = port_get_link_settings,
        .set_link_ksettings = port_set_link_settings,
        .get_link = ethtool_op_get_link,
        .get_strings = port_get_strings,
        .get_ethtool_stats = port_get_ethtool_stats,
        .get_ethtool_stats_clear = port_get_ethtool_stats_clear,
        .get_sset_count = port_get_sset_count,
        .set_phys_id = port_set_phys_id,
        .get_drvinfo = sx_get_drvinfo,
        .get_module_info = port_get_module_info,
        .get_module_eeprom = port_get_module_eeprom,
        .get_fecparam = port_get_fecparam,
        .set_fecparam = port_set_fecparam,
        .get_ts_info = sx_get_ts_info,
};
```

Listing 6: Ethtool Operations implemented in `sx_netdev`

5.2.4 Missing Offload Flags

A few netdevice offload flags and operations were missing from the original driver provided by Mellanox. It is likely that the Mellanox OS does not rely on them as it does not use the Linux networking stack but its own custom networking stack although running on a Linux kernel.

The following missing features were added to `sx_netdev` netdevices:

- NETIF_F_HW_SWITCH_OFFLOAD: this flag indicates that the netdevice will offload kernel entries to a hardware switch and should be notified when joining/leaving a e.g., a bond or a bridge;
- ndo_change_proto_down: this netdevice operation changes the link status of the device to `PROTO_DOWN`, used when a pair of CLAG nodes disable a particular link between them which is not needed, being a redundant link.
- offload_fwd_mark: this flag is set on all unicast packets received on a `sx_netdev` device to prevent the kernel bridge driver from forwarding the packet back to the network as it was already done in hardware.

5.2.5 Default VLAN Tagging in Mellanox OS

After debugging a case where ARP packets coming to the switch CPU were dropped in the `arp_rcv` kernel function, we discovered that the `sx_netdev` driver would add a default VLAN tag to all received untagged packets. The Linux kernel, not knowing which VLAN this tag corresponds to, would simply drop the packet. The purpose of the default VLAN tag is unknown but it is likely that the Mellanox OS has an internal use for it. A wireshark screenshot of the culprit is shown on Figure 20. The fix was simple: strip the VLAN tag before passing the packet to the network stack.

5.2.6 Driver and SDK reloading

One of the main missing `sx_netdev` requirement was the ability to reset the switch ASIC and reprogram its configuration on a switchd restart but while keeping the same netdevices. This is

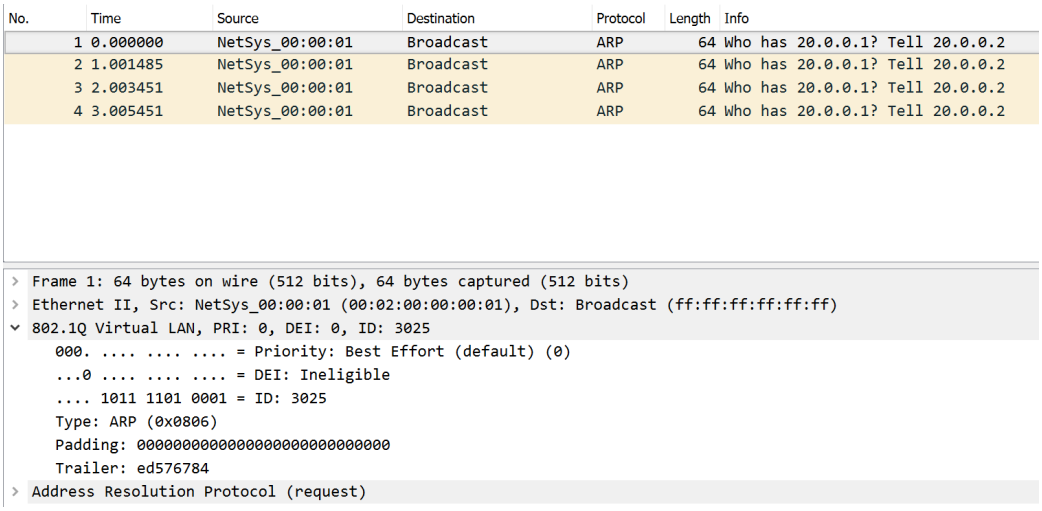

Figure 20: Default VLAN tag for untagged frames (ARP packets in this example)

required as any network construct such as a bridge or a bond created before the reinitialization will still reference the same netdevices. However, when reloading the SDK and resetting the chip, the `sx_netdev` driver would get reloaded as well and the netdevices would be deleted. A feature request was sent to Mellanox which provided a driver patch to enable chip reset as well as reloading of all drivers except `sx_netdev`. After a few configuration tweaks (the `sx_netdev` driver module would depend on the ASIC driver and the latter gets reloaded, Mellanox provided an additional fix to break the dependency), the driver reloading was made to work.

5.2.7 Build System Refactoring

The building system used originally at Cumulus Linux for the Mellanox kernel modules worked in the following way:

1. All sources are extracted to a temporary build directory;
2. Custom Cumulus modification are applied via a series of patches
3. Additional back-porting patches from Mellanox are applied last
4. Make is invoked and the SDK compilation ends with building installable Debian packages

As the `sx_netdev` was modified extensively in this work, complex patch conflicts started to appear between the Cumulus patches (point (2) above) and the back-porting patches (point (3) above). We refactored the building system to apply the back-porting patches first and the custom Cumulus patches last.

5.3 Evaluation

Ensuring correctness of the new driver was the first milestone for this implementation. The new `sx_netdev` driver was tested against an internal *smoke* test suite which includes among

others L2, L3 and ACLs testing. Each test simulates a network topology with soft-nodes (*kvm* nodes) for hosts (as well as additional routers/switches) and one hard-node, the device under test (DUT), being an actual Mellanox switch. The `sx_netdev` implementation passed all *smoke* tests.

The performance of this new driver was measured second, expecting higher performance than the TUN/TAP scheme. We measured the throughput of control path processing using `iperf` between a soft-node and a Mellanox SN2700 switch. The control path processing performance was measured to be unchanged from the TUN/TAP. Additional debugging is needed to figure out what limits the performance increase as the equivalent Broadcom driver experienced a 4× speedup for the receive path (and 2× speedup for the transmit path) compared the the TUN/TAP scheme. The benchmarks for the TUN/TAP driver are shown in table 1. For the `sx_netdev` driver, they are shown in table 2.

MTU	**1500**	**9000**
RX (Mbps)	12	72
TX (Mbps)	69	189

Table 1: TCP Throughput Performance, TUN/TAP Driver

MTU	**1500**	**9000**
RX (Mbps)	12	69
TX (Mbps)	73	192

Table 2: TCP Throughput Performance, `sx_netdev` Driver

5.4 Conclusion

The modified `sx_netdev` driver together with the new infrastructure based around the `sx_netdev` kernel-based was merged to the main development branch in January 2018 and deployed to all testing Mellanox switches. A few bugs were found during QA testing but overall the merge did not encounter many issues. It will be included in the next release, Cumulus Linux 3.6.0, targeted for the end of April 2018.

6 Switch Port Configuration

In the switchdev model, the switch ports are exposed as `net_devices`. Users configure these ports using standard Linux networking tools such as `ethtool` or the `ip` tool from the iproute2 suite. In the initial TUN/TAP model used by Cumulus Linux, TUN netdevices are created for each physical port by switchd, the user-space switch daemon. Port configuration through the Linux networking tools is forwarded back to the user-space SDK from the kernel to actually configure the ports as the TUN interfaces were only placeholders for actual netdevice ports. Furthermore, the general port configuration scheme is divided between kernel and private user-space parts in switchd. In order to standardize creation and configuration of switch ports as a built-in kernel construct, a new switch port configuration scheme was developed. This section describes this contribution by detailing the different stages involved and which interfaces was chosen to implement them.

6.1 Port Creation

As the switch ports are exposed as `net_devices`, the netdevice driver needs to know the number and types of ports in order to create them. In the TUN/TAP model, the port topology is given through a user-space configuration file and switchd creates TUN netdevices according to the ports listed in the configuration file (including split ports). However, Cumulus Linux' goal is to have a common interface between the operating system and the switch hardware. This interface would enable the kernel to use standardized functions to manage and retrieve information about switches. Using APD (see side-note below), the kernel can query at boot-time the number and types of ports available on the switch ASIC. The network driver can thus create one `net_device` per port without requiring platform information stored in user-space configuration files.

ACPI and APD
ACPI: Advanced Configuration and Power Interface (ACPI) is an open-standard wrapping BIOS/UEFI functions to be used by operating systems to discover and configure devices.
APD: ACPI Platform Description (APD) is an extension of the ACPI standard for networking devices. Hardware vendors can store networking platform specifications, e.g. port types, topology, ASIC version in BIOS firmware and this way allow the OS to query this information using a standard interface [9].

6.2 Port Ganging/Splitting

Switch ports can be combined or split into sub-ports. Combining ports to create LAG interfaces is done through the Linux bond driver. The bond configuration is offloaded to hardware is done through the SDK's dataplane configuration. For packets coming to CPU, bonding is handled in software by the bonding driver in the kernel.

Splitting ports (e.g., split`(swp1) -> swp1s0, swp1s1, swp1s2, swp1s3`) requires the original `net_device` to be deleted and then to create the split sub-interfaces. For some specific vendors, splitting a port can disable the following port (as the hardware might use the next port for the split). Therefore splitting ports is a hardware-specific operations that should be handled by the vendor's network driver. In the TUN/TAP model, the split port netdevices were created directly with their corresponding names by switchd. The new kernel switch model should allow the user to split/unsplit an interface from the command line or using a kernel-user interface.

The chosen interface to split and unsplit port is through the `devlink` port configuration interface using the following command:

```
$ devlink port split DEV/PORT_INDEX count COUNT
```

Ideally, splitting ports should be performed by detecting split/unsplit cables directly from hardware (or firmware) and updating the port configuration accordingly without user-space intervention. Devlink was however chosen for as a first milestone. `phylink` is a kernel module in development which can detect cable events from the kernel and trigger a split/unsplit. When mature enough, it will replace the currently used `portwd` user-space tool used to inform the user-space switch daemon of cable events.

Devlink
Devlink is a recent generic Netlink interface proposed by Mellanox to expose switch-ASIC-specific information and operations that should not be in the `net_device` structure.
Use cases:

- get/set of port type (Ethernet/InfiniBand)
- monitoring of hardware messages to and from the ASIC
- setting up port splitters - split port into multiple ones or squash them again, enabling usage of splitter cable
- setting up shared buffers - shared among multiple ports within one chip
- monitor hardware resources usage and memory/table hierarchy

The devlink command invokes devlink functions, which are part of the iproute2 package, functions which are also exposed as a user-space API. These user-space functions then call the devlink kernel API (through the Netlink user-kernel interface). A switch network driver implements devlink operations by implementing and registering the `devlink_ops` structure. As an example, the `port_split` function in the `devlink_ops` structure, is the function to implement to split ports. Its arguments are the devlink instance, the port index and the final number of ports.

6.3 Setting speed, duplex and auto-negotiation

`Ethtool` is the Linux utility tool to set speed, duplex mode and auto-negotiation of network interfaces. Every vendor network driver has to implement the ethtool set of operations. In the TUN/TAP model and for the trampoline driver case, the ethtool operations are forwarded back to the user-space SDK.

Ethtool is a Linux tool to control and change network devices settings, especially Ethernet devices. It is also used to retrieve stats from network interfaces and query hardware time-stamping capabilities.
A network driver implements ethtool operations by implementing and registering the `ethtool_ops` structure.

6.4 Setting MAC addresses

The final step is to assign a valid MAC address to every interface. These MAC addresses are provided by the vendors and read from the system EEPROM. Currently, `switchd` reads the system EEPROM using a user-space python utility. This will be moved to the network driver, reading the MAC address with APD and assigning it directly to the interface at creation-time.

6.5 Evaluation and Future Work

Except setting MAC addresses, all the steps mentioned above have been implemented and will be integrated in the next major Cumulus Linux release (Cumulus Linux 4.0). The `sx_netdev` driver, presented in section 5, was extended to:

1. retrieve information about switch ports with APD and create ports at boot-time; a simplified APD interface, consisting of only the port index and the port type (e.g., SFP+, QSFP+) is stored in firmware;
2. implement the devlink interface and operations such as port split/unsplit; and
3. forward ethtool operations to the SDK via a private Netlink channel.

The same work was done for the Broadcom network driver, `knet`. The MAC addresses are still set later in the initialization process by switchd as the kernel's EEPROM utilities need to be forward-ported first to the newer Linux kernel version used for Cumulus Linux 4.0.

7 Resource Management

Managing ASIC's limited hardware resources and dynamic partitioning (shared hardware resources for routes, FDB entries, etc.) is already a challenge by itself. The Linux kernel has no constructs to represent them and query their current usage level. The kernel should be able to notify user-space daemons that resources are exhausted and give hints on how particular resources should be managed (e.g., prefer routes over MAC address in L3-heavy environment). This resource management challenge is discussed in a paper written for (and presented in) the Netdev 2.2 conference (Technical Conference on Linux Networking) [24]. The content of this section is the aforementioned paper followed by an evaluation section reviewing the results and discussing the future steps that will be taken for resource management.

Abstract

The Linux kernel supports offloading of networking functions such as bridging and routing to switches (and NICs). These offloads occupy hardware resources that are difficult to track for the kernel as their representations can differ from kernel resources. In addition, hardware resources must be often traded off against each other if they are stored in the same shared memory.

Failure to offload a kernel resource, e.g., a routing entry, installed in the kernel puts the kernel and hardware device out of sync which can result in incorrect behavior, e.g., blackholed packets. Reverting to a software implementation is not always possible, especially if line rate processing of packets is a hard requirement of the network environment.

This section presents the challenges of resource management involved in network function acceleration. We present models to manage complex resources offloads where kernel/hardware representation do not match. Resource availability is checked synchronously on the offload path to predict with a high degree of confidence that an offload will succeed while also providing synchronous feedback to users, e.g., routing daemons.

7.1 Introduction

The Linux kernel supports accelerating network functions by offloading to hardware. Typically, stateless decisions are offloaded, and more recently enhancements like FDB offload, FIB offload and eBPF offload support have been added. The situation however gets a little more complicated when offloading functions that affect multiple interfaces and need to be optimized and managed across these multiple interfaces. Further complications arise from mismatch in the view of resources between the kernel and hardware/driver. When offloaded to hardware, a single kernel resource may occupy more than one hardware resource: e.g. routing prefixes which are a complete entry in the kernel might be stored in TCAM while next-hops could be stored in a hash-based memory.

One important implementation challenge of hardware offloading is to keep the kernel state in sync with hardware when a network function cannot be offloaded due to lack of hardware resources. Let us consider the L3 routing case and how hardware offload failures can cause catastrophic problems. Failure to indicate a hardware offload error to a routing daemon, will result in the routing daemon advertising reachability of the failed prefix, which will result in peer routers sending traffic towards it, which will then get blackholed. The device driver may try to remedy

the situation by disabling offloading and revert to software forwarding but this solution is not always practical and can actually even be harmful if forwarding packets at line rate is a strict requirement of the network infrastructure. A data center switch such as the Mellanox Spectrum ASIC can forward packets at 6.4Tb/s while software forwarding on the same ASIC delivers around 4Gb/s [18]. With such a mismatch in speeds many packets will be lost as the CPU cannot handle line rate traffic.

As said previously, hardware offloading failures can also result in incorrect route advertisements and potentially blackholing of traffic. Consider the network topology shown on Figure 21 with 3 routers and a Switch ASIC (DUT) acting as an additional router. All routers running BGP. Suppose the hardware FIB on the switch has a default route, e.g., 0.0.0.0/0, for packets and a new route, 13.0.0.0/24, is installed but its offloading fails without notifying the kernel or the routing daemon (assume no fallback to software forwarding). Packets for 13.0.0.0/24 will still be forwarded in hardware but will take the default route instead of the proper next-hop which will result in incorrect routing and blackholed packets. Details about this experiment can be found on Github [23].

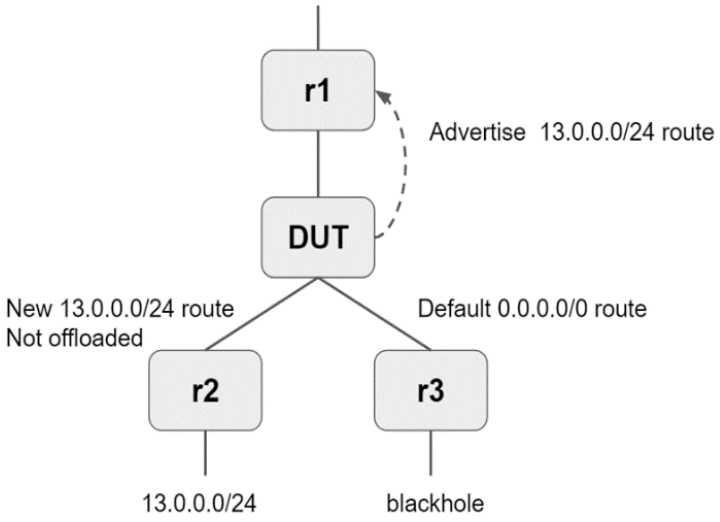

Figure 21: Lost packets if a route is not offloaded

Hardware implementation can differ drastically from the software implementation both in terms of processing and resource management. For example, IPv6/64 and IPv6/128 prefixes (routes with a /64 mask versus a /128 mask) typically use different hardware entries while the kernel does not distinguish between them. Hardware representation mismatches make it harder for the kernel to keep a precise and simple accounting of hardware resource occupancy. This is further complicated as different kernel resources can be mapped to the same shared hardware resources: e.g., if both routes and neighbor entries use the same shared memory on a device then adding more routes can decrease the number of available neighbor entries. This leads to problems where an IPv6 route add could result in decrementing the IPv4 address pool, which is an unexpected result.

Throughout this section, we use examples of Linux kernel networking functions offloaded to a switching ASIC. The main network function used to explain the problems and solutions is L3

routing. However, the same principles and problems apply for other network functions as well as other devices such as NICs.

The first three subsections provide background information on switch ASIC pipelines, Linux kernel hardware acceleration and the current approaches to manage resources in the kernel. We then describe possible resource managers design to improve user feedback and avoid unhandled resource overload.

7.2 Switch ASIC Pipeline Resources

Switch ASICs have different pipelines and resources, however main components can be abstracted as shown on Figure 22 and Figure 23 (showing respectively a packet's ingress and egress pipelines).

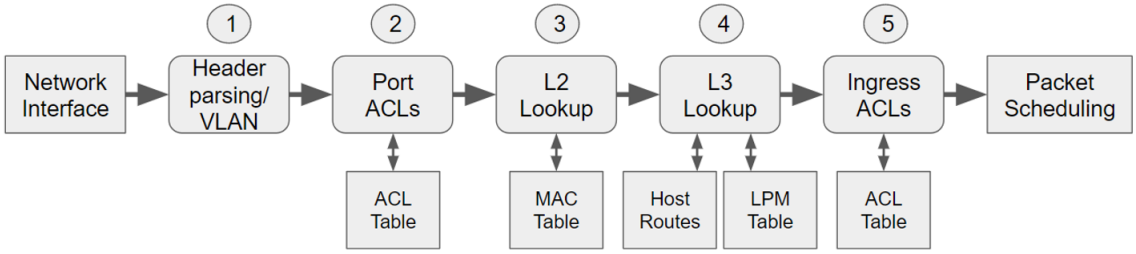

Figure 22: Switch ASIC Ingress Pipeline

We discuss first the ingress path (Figure 22) with a focus on hardware resources used:

- **Port ACLs (2)**: filtering of packets before forwarding, based on source port and VLAN; ACLs usually stored in TCAM.
- **MAC Address Table (3)**: exact-match table for (VLAN, destination MAC) tuples. It decides if a packet must be routed or bridged. In case of bridging, the table contains the egress physical port; usually stored in hash-based memory. Packets that match the "router MAC" are deemed to be routed.
- **Host Routes Table (4)**: Exact match for (VRF, IP) tuples. Given a destination IP address, it provides the directly connected port and next-hops which are /32 routes (or /128 for IPv6); usually stored in TCAM or hash-based memory.
- **LPM Table (4)**: Longest-Prefix Match on IP addresses for (VRF, destination IP). It is used mainly for remote routes as well as connected routes that have not been resolved and placed in the host table; usually stored in hash-based memory on modern silicon implementations.
- **Ingress ACLs (5)**: filtering of packets after forwarding, based on L2/L3/L4 fields; ACLs usually stored in TCAM

After having been buffered and scheduled, a packet goes through the egress pipeline, shown in Figure 23. The packet's header is being rewritten with new needed fields, e.g., MAC address, VLAN id might be added and egress ACLs are applied as egress filtering. Egress ACLs are stored

usually in TCAM as well. Some ASIC architectures manage all different ACLs attach points (port, ingress and egress) through the same shared hardware table and thus induce tradeoffs on the overall ACL capacity.

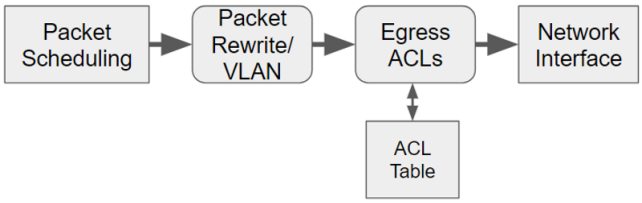

Figure 23: Switch ASIC Egress Pipeline

Shared Hardware Resources

Switch ASICs resources are often shared between different types of objects. For example, the host routes table and MAC address table might use the same underlying memory, typically implemented with some form of a hash table. The entries and keys that use the table are defined in blocks that can be resized and thus allow routes versus mac capacity to be traded off against each other. Furthermore, different types of host route entries and LPM entries exist (e.g., IPv4 entries, IPv6/64 and IPv6/128 entries) which can also share the same memory and be traded off against each other.

7.3 Linux Kernel Hardware Acceleration

This section provides a survey of network function elements currently being offloaded to a switchdev driver in the Linux kernel. Resources used in the offload are described as well as the current resource management scheme and the impacts of hardware resource overload.

Address Lists offloaded to NIC e-switch

Unicast and multicast address lists can be offloaded to SR-IOV enabled NICs, e.g., ixgbe, for filtering and switching in hardware. MAC addresses are added through the netdev op `ndo_fdb_add`. Address lists can be directly offloaded to hardware without adding to the kernel (kernel bypass via `NTF_SELF`).

Drivers check the hardware limits before programming the address to the hardware and returns `-ENOMEM` when no resources are available and fall back to software instead. The cpu complexes in systems on which this is deployed are usually capable of handling line rate traffic therefore falling back to software is a viable option.

L2 Bridging

Offloading of FDB and MDB entries enables switching of packets at line rate based on destination MAC addresses. If learning is done in hardware, hardware learnt entries are pushed to kernel

FDB. Software learnt or managed FDB entries are offloaded to hardware through via switchdev notifiers.

In case of failure due to the absence of an FDB/MDB entry, hardware or kernel will flood the packet to all ports in the broadcast domain and functionality is still preserved. While this preserves functionality flooding is suboptimal in a L2 domain as it takes up unnecessary network bandwidth and CPU cycles.

L3 Routing

Hardware drivers learn of changes to Layer 3 configuration via notifiers. For example, when a network interface is enslaved to a VRF, a netdev notifier is called or if an address is added or removed on a network interface an address notifier is called. The driver also uses that notifier to check on the fly if any additional hardware resources are needed with these interfaces. For example, a router interface (RIF) might need to be allocated in hardware or a virtual router id needs to be assigned for a VRF. If the maximum number of such resources has been reached, the driver can synchronously fail the request and the error is returned to the user.

Similarly, switchdev drivers learn of changes to kernel FIB entries (adds, deletes, and modifications) via FIB notifiers. Because of the potential overhead in offloading FIB entries, the work needed to program the change in hardware is done asynchronously from the user request (e.g., via a work queue). This means the change is made to the kernel FIB, and assuming no errors adding the work to the queue, the kernel returns success to the user which can continue with more FIB changes. Programming the hardware is done later which means if hardware capacity is exceeded there is no way to pass an error back to the user. Since the FIB entry exists in the kernel but not in hardware, the two forwarding paths are now out of sync. One response to such overload scenarios is to abort all FIB offloads and fall back to software based forwarding. As described earlier this situation simply does not always result in acceptable system behavior.

ACL Offload

ACL's are rules that specify whether traffic matching the criteria is forwarded or discarded. In Linux ACL's can be configured via netfilter or tc. An API to offload tc rules to hardware is a netdev operation named `ndo_setup_tc`. Drivers implement this netdev operation if they can support tc rule offload. In most cases hardware is not capable of offloading all ACL rules that software can support. For example, the driver can choose to offload flower and u32 classifiers and not the many others that tc can support in software. The Linux kernel does not fail an acl offload if the hardware does not support the ACL today. For a switch ASIC this means, on an ACL offload failure, the ACL is active only on the packets punted to the CPU and not on the packets forwarded in hardware resulting in inconsistent and unpredictable ACL enforcement. In environments where strict security policies are audited and signed off on, this would be a violation.

Offload Policies

Having seen the previous examples, offload policies in the kernel can be generally classified in three different models:

- **Sync Model**: hardware and kernel state are kept in sync. In case of insufficient hardware resources, fall back entirely to software, e.g., routing, ACL's
- **Bypass Model**: hardware offload is managed separately from the kernel and skips the kernel, e.g., address filters on NICs.
- **Hybrid Model**: Partial offload to hardware either through user flags (`skip_sw`, `skip_hw`) or when no more hardware resources are available, fall back to software for additional entries, e.g., ACLs.

This classification is not consistently respected in the kernel and each network function develops its own custom resource management. It also sometimes differs between drivers for the same function. While each of the solutions and approaches works for building solutions to specific problems, there is no consistent platform expectation that a user-space process can have that spans all solutions.

7.4 Approaches for Resource Management

Driver Resource Profiles

Driver resource profiles partition the available memories in advance between the different hardware entries. The partitioning guarantees a given number of entries (e.g., 50K MAC addresses) and is thus conservative: the unused space for MAC addresses cannot be used for, e.g., more IPv4 routes. Moreover, resource overruns are left as an exercise for the consumer of those resources and there is no simple mechanism that prevents a user space daemon from oversubscribing. As we have noted in the earlier sections, this can lead to catastrophic and undetectable networking failures.

Simple accounting for Dedicated Resources

Resources like RIFs or VRFs have static limits where the device has guaranteed support for a specific number of such resources. They are more easily tracked for example using simple counters and allow for synchronous failures when the limit is reached.

Try and Abort for Complex Resources

FIB entries, MAC entries and next-hops share underlying memories and the kernel representation can be different from the underlying device representation (e..g, IPv6/64 and IPv6/128 addresses use different hardware entries). If driver resource profile is not used for these resources (ie., guaranteed limits) and with no means for querying current resource utilization, user-space has no way to know if FIB changes need to be limited. A routing daemon can only push the request to the kernel and hope for the best – that the change does not exceed hardware capacity and trigger an offload abort.

7.5 Problem Definition

The introduction and background sections motivated the need for:

- a clear offload failure error path to the user or protocol daemon;
- better communication between kernel and drivers/hardware to account for resource usage and availability
- A resource manager model or resource management algorithm for drivers. A resource manager allows shared memories to be used more efficiently and with more flexibility than the strict partitioning of resources used in driver profiles.

We discuss in the rest of the paper different models for drivers to manage hardware resources. The main goal of these models is to fail synchronously to the protocol daemons in case of insufficient resources.

Considering the L3 lookup case again, three main loops are involved in a typical workflow as shown in Figure 24. Loop (1) represents users, e.g., routing protocol daemons, adding routes. The kernel checks with the device driver that the device still has sufficient resources for a route in Loop (2), one route at a time. Resources are only reserved at that point as drivers typically defer the actual offload to the device through a work queue for latency reasons. Loop (3) represents the driver reserving hardware resources through communication with the device (PCIe reads/writes).

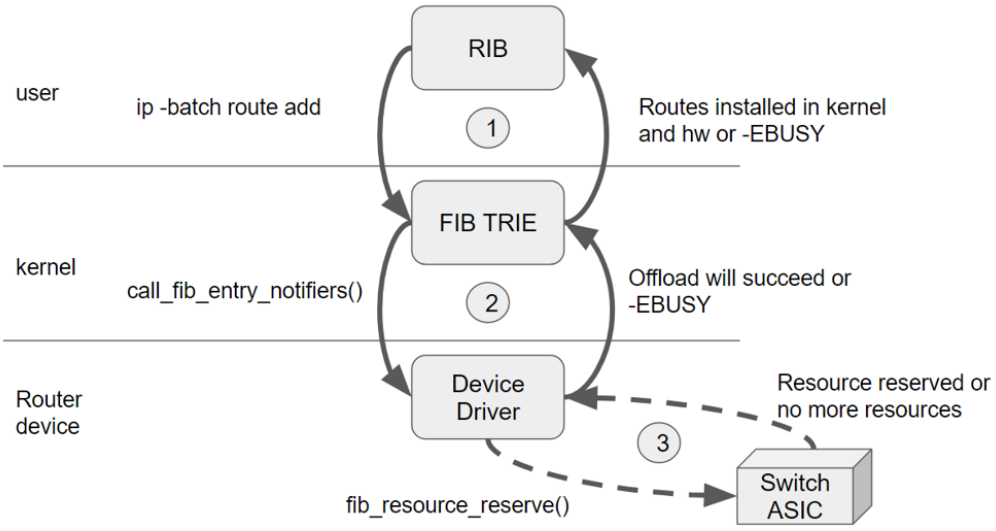

Figure 24: Route Add Workflow and Error Path

Loop (1) and (2) are synchronous, (1) representing the batch add of routes through rtnetlink while (2) is reserving resources needed one route at a time. Loop (3) can take three different forms:

1. synchronous, install route synchronously: **lockstep implementation**
2. synchronous, prefetch resources for future routes: **synchronous prefetching**

3. asynchronous, prefetch resources for future routes: **credit-based implementation**

The synchronous error flow in case of insufficient resources is described in more details below:

Loop (3): If there are not enough hardware resources to offload a kernel resource, the device firmware will return a driver-specific error to the device driver. Reserving resources for an IPv4 route can fail, e.g., because there is no more IPv4 prefix entries available on the device.

Loop (2): Reserving hardware resources for a given offload thus fails and the driver returns `-EBUSY` to the kernel core. In the routing case, `fib_table_insert` cannot reserve needed hardware resources on devices and does not install the route in the kernel. Routes that have been previously offloaded as part of the same rtnetlink batch are removed for consistency. **Loop (1)**: Routing daemons receive the `errno` indicating no more resources available on the device. If a batch add fails, the entire batch fails and no resources are actually offloaded.

An alternative solution that gets rid of Loop (3) is presented as the **predictive solution**.

Lockstep Implementation

In the lockstep solution, routes are programmed synchronously to the hardware when right after they are added to the kernel. This solution introduces latency (PCIe reads/writes to configure resources in the hardware) on the control path for installing a route. We measured the latency on a Mellanox Spectrum ASIC (SN2100) to be **50-55**μs for install of an IPv4 route with a single nexthop. When installing many routes, this additional latency increases route convergence time, e.g., if a link goes down and routes need to be recomputed and added to the kernel and hardware. In case of offload failure, routes are removed from the kernel to keep hardware/kernel states in sync and an error is returned synchronously to the user.

```
1 inet_rtm_newroute() // called via rtnetlink
2         fib_table_insert()
3                 router_fib4_insert()
4                 // program hardware with new route
5                 // fail synchronously in case of offload failure
```

Listing 7: Lockstep Workflow

Patch to the spectrum driver used to measure the latency when installing a route inline can be found on Github [21].

Synchronous Prefetching

With synchronous prefetching, the driver reserves N hardware route resources in advance. The next $(N-1)$ offloads will be able to reserve routes and return immediately without any hardware access. Routes are installed to hardware asynchronously (route install is deferred using a work queue) but the offload will succeed as resources have been reserved. The critical path of installing a route still incurs the additional I/O latency every N routes (worst-case scenario) which can still impact route convergence time. Thus, the prefetching should be removed from the control path which we discuss in the next section.

```
1 inet_rtm_newroute() // called via rtnetlink
2          fib_table_insert()
3                 router_fib4_prefetch()
4                // prefetches synchronously new
5                // resources if needed, returns immediately o/w
```

Listing 8: Synchronous Prefetching Workflow

Credit-based Solution

The credit-based solution hides the latency of the synchronous solution while still providing an immediate feedback to the user. The driver periodically refills, on the side, a "bucket" of N route resources. This allows for modeling the hardware's capacity and rate of intake accurately which is yet another error that can happen on the hardware front. The size of the bucket can be adaptive: starting with a bigger bucket size and progressively reducing its size as hardware resources decrease. The driver refills the bucket asynchronously when it reaches a low threshold such that it can always reply synchronously and without any involving hardware access to a route add request.

```
1 inet_rtm_newroute() // called via rtnetlink
2          fib_table_insert()
3                 router_fib4_reserve()
4                 // triggers async refill if bucket reaches
5                 // low threshold
6
7 router_fib4_refill()
8                // refills asynchronously bucket with new
9                // reserved resources
```

Listing 9: Synchronous Prefetching Workflow

We simulated this implementation (hardware accesses to reserve resources are simulated) with the Mellanox switchdev driver (`mlxsw`) to show that the additional latency from the lockstep implementation is hidden. The source code for the credit-based solution can be found on Github [22].

7.6 Comparing Lockstep & Credit-based

We compared the performance of the lockstep implementation and the credit-based solution for FIB updates using two metrics: coherent control plane latency and dataplane latency. We used a Mellanox SN2100 Open Ethernet Switch (Spectrum ASIC) with net-next kernel. The Spectrum driver was modified to install routes inline (lockstep implementation) and alternatively to use a

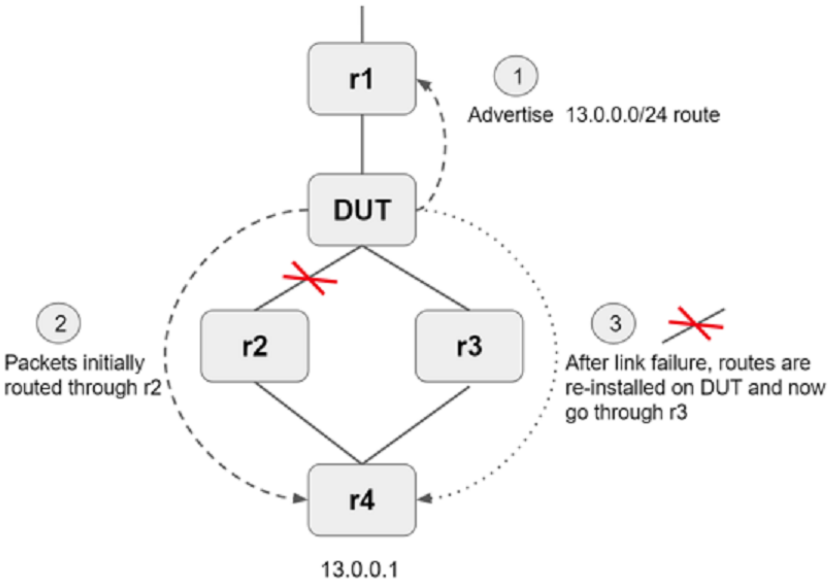

Figure 25: Setup for 10,000 routes being updated from FRR to hardware

bucket refill system to hide the lockstep latency (credit-based solution). The experimental setup is shown in Figure 25.

The coherent control plane latency represents the round-trip time (from user-space and back to user-space) to install new routes in the kernel. It is coherent as it includes the time to update hardware tables as well in order for the kernel and hardware to be in sync at the end of the updates.

We measured coherent control plane latency using the Unix `time` command (focusing on system time) when installing 10,000 routes with `ip -batch add`. We checked that the results are coherent using perf tracepoints measuring hardware access latency. For the lockstep implementation, we measured the control plane latency to be **600ms**, which gives a latency per route of **60us**; the credit-based system install the 10,000 routes in **480ms**, giving a speedup of **1.25**.

We also measure dataplane latency, i.e., time for packets to start flowing after 10,000 routes are being installed. We measured the worst-case latency, i.e., when the last route being installed is the one needed for the packets to start being routed. We ran `ping` with an interval of 0.01s between packets and look at the icmp sequence number of the first packet successfully transmitted. Route installation is started at the same time as the `ping` command. We measured dataplane latency to be **1.10s** when installing 10,000 routes (110 icmp packets were lost) for the lockstep implementation.

With the credit-based solution, dataplane latency goes down to **0.81s** (80 icmp packets lost) when installing the same 10,000 routes. Packets thus start flowing **1.3**× faster when using the credit-based solution.

The results are summarized in Figure 26:

- **Users Latency** represents the consistent control plane latency;
- **HW Latency** is the constant latency to access hardware when installing a route, hidden in the credit-based implementation.
- **Query Latency** measures the actual time to query hardware, increased by 10× in the experiments (up to 350us) to show that the credit-based solution would not suffer if querying hardware about resources was costly.
- **Data Outage** shows the dataplane latency.

The setup for the experiments (including BGP configuration) can be found on Github [22].

Method	Users Latency	HW Latency	Query Latency	Data Outage
Lock Step	600ms	60us	35us	1.10s
Credit based	480ms	60us	35us	0.81s
Lock Step	600ms	60us	350us	3.98s
Credit based	480ms	60us	350us	0.84s

Figure 26: Measurements for 10,000 routes being updated from FRR to hardware

7.7 Predictive Solution

The predictive solution computes the current occupancy of hardware resources (overestimates if necessary) and predicts with a high degree of confidence that an offload will succeed.

Flat Model: Simple, Hard Limits for Resources

This solution is equivalent to driver resource profiles: simplistic assumptions on capacity and usage for each object type are chosen which allow for fast in-line checks of a configuration change with a high degree of confidence that hardware offload will succeed. This model always works but never pushes hardware to full utilization because of the simplifying and conservative assumptions.

Complex Model of Resources and Usage

With enough details about the hardware, a more complex resource usage predictor can be built: kernel resources are mapped to their corresponding hardware resources. The dependencies between hardware tables and shared memories is known by the driver which can then compute the

current usage and remaining entries given the current kernel state. Current kernel accounting/-models of resources could be modified to better align with typical hardware resources and ease the hardware resource accounting.

7.8 Related Work

The DPIPE interface [20] provides visibility for the user into the hardware pipeline and [19] extends this interface to provide information about available hardware memories, including shared memories and memories' current usage. [18] presented the same issues involved in hardware overload and discussed the need for synchronous feedback in case of hardware failure.

7.9 Future Work

Feedback from the conference will help confirm and direct the work on a resource manager. The credit-based solution will be extended and studied in more details to model rate mismatches from user down to hardware as well as resource representation mismatch. We expect higher speedups with the credit-based solution than found in this paper (using a workqueue item for the refill thread is not the most efficient).

In addition, kernel data structures and representations of offloaded data can be improved to better align with hardware, for example to simplify accounting.

7.10 Conclusion

Hardware acceleration of network functions offloads kernel resources on hardware devices. These kernel resources occupy hardware resources and the current expectations in case of insufficient hardware resources vary from cases to cases. For L3 routing, the expectation is fall back to the software implementation but reverting to software can be harmful if line rate processing of packets is required for a given network infrastructure. Other network functions such as ACLs, address lists on NICs and L2 bridging develop their own custom solutions for resource overload which similarly might not be efficient enough and do not always provide synchronous feedback to the user.

This section motivated the need for synchronous feedback to the user in case of offload failure in order to avoid incorrect or harmful function behaviors. Different models were presented to deal with the representation mismatch between hardware and kernel resources. A credit-based solution with hardware resources prefetching helps managing complex resources involved in, e.g., routing, where routing resources end up in shared hardware resources and have to be traded against other resources. A predictive model is harder to engineer if resources are shared and vary greatly between devices but can be used in simpler instances.

7.11 Evaluation & Future Work[1]

The Netdev 2.2 paper about resource management, presented in this section, evaluated different resource management schemes: lockstep implementation, synchronous prefetching and credit-

[1] This subsection is not part of the original Netdev paper

based implementation. It showed that the credit-based solution will hide prefetching latency, allowing fast resource acquisition and synchronous feedback to user-space daemons.

The hardware interaction was simulated using time delays in the benchmarks as the kernel does not currently have an internal resource management API. However, this research led to a better understanding of the interfaces and functions needed to perform resource management, including interactions and interfaces between the networking part of kernel and the drivers. The switch ASIC drivers will likely implement a standard set of resource management operations, allowing, for example, the kernel to query available resources and confirm to user-space daemons that the switch ASIC has enough available to implement the desired data plane configuration. This resource management API is in its design phase: it is currently discussed between Linux developers contributing to the corresponding part of the kernel.

As a first step, a static partitioning of resources, made available to user-space daemons through sysctl entries (e.g., `ipv4/route/max_size` for maximum number of routes available in hardware) is under development at Cumulus Networks. Cumulus Linux users did encounter the issues presented in the Netdev 2.2 paper, issues due to exhausted hardware resources. Being able to query the maximum and currently available number of given resources will help customers and applications diagnose and troubleshoot issues due to switch ASIC resources in a standard way. These numbers are set, also through the sysctl interface, by the user-space SDK. The limits are then enforced in the kernel when users (e.g., protocol daemons), try for example to insert a new route or ACL entry, ensuring that the hardware capacities will not be exceeded.

8 Related Work

In this section, we relate concepts and designs done in this thesis to other research and work done in the networking field, with a focus on Linux-based networking as well.

8.1 Offloading Network Functions in Linux

Offloading networking functions for higher performance is not a new idea in the Linux kernel. Modern high-end NICs come, for example, with TCP/IP checksum computation, packet segmentation and VLAN tagging offload capabilities. The Linux kernel has support for offloading these functions (specific offloading support can be checked on a particular NIC with `ethtool -k`) [34].

More advanced NICs come with embedded switching capabilities such as the Mellanox ConnectX-4 series of NICs [35]. These NICs can perform switching directly in hardware between virtual NICs dedicated to VMs thus greatly increasing network performance between in the virtual network environment. Mellanox added support for embedded switching offloading to the Linux kernel for the ConnectX-4 series of NICs.

Encryption offloading is another important function that can be offloaded. 4.13 was released with in-kernel TLS implementation, bypassing the previous user-space TLS encryption method. With encryption now performed in-kernel, a standard in-kernel TLS offloading API can be designed, which was done in [36].

8.2 Resource Management in ASICs

The DPIPE interface [20] provides visibility for the user into the hardware pipeline and [19] extends this interface to provide information about available hardware memories, including shared memories and memories' current usage. [18] presented the same issues involved in hardware overload and discussed the need for synchronous feedback in case of hardware failure.

9 Conclusion

Modern data-center network operating systems rely on proprietary user-space daemons wrapping SDKs from switch vendors. Linux-based variants of these operating systems either use an out-of-band networking stack (Mellanox OS or Cisco's NX-OS [16]) or work around the lack and limitations of offloading constructs in the Linux kernel (Cumulus Linux). Fortunately, the future is brighter for Linux and network switches: hardware-accelerated network functions are becoming of first-class citizen of the Linux networking stack and the Linux kernel community is ready to accept changes needed to include control- and data-plane offloading support. Linux would then be a complete network operating system with accelerated network functions.

This thesis explored the different requirements to support switch ASICs in the Linux kernel through the switchdev model of exposing switch physical ports as Linux netdevices with network drivers, using standard Linux networking constructs such as bridges and routes and offload them to hardware via a work-in-progress data-plane offloading API. A Mellanox network driver was ported to Cumulus Linux to accelerate and take control plane packet processing and out of the users-pace switching daemon and SDK path. Creating switch port netdevices is moved to boot-time platform information stored in BIOS/UEFI while the interfaces configuration is done from the netdevice driver and a few user-space tools (devlink, ethtool) rather than almost entirely from the vendor's SDK. In order to support user-space SDKs from the in-kernel switchdev model, a trampoline driver forwards switchdev calls back to user-space.

Finally, the resource management challenge was discussed. The kernel needs to be aware of the available switch ASIC resources. At the very least, it must to be able to return an error for insufficient resources to routing daemon. Failure to offload routes because of insufficient hardware resources can lead to black-holed packets while falling back to software forwarding is strictly not an option for high-end data-center networking. A credit-based solution was presented which hides the PCI latency to query the switch ASIC about currently available hardware resources. The main blocker for that solution is the on-boarding of switch vendors to provide such an API to query the ASIC for hardware information from the Linux kernel itself (and not the SDK). The trampoline driver proves itself useful in this case as well, by forwarding switchdev resource management API calls back to the user-space SDKs.

Acronyms

ACL	Access Control List
ASIC	Application-Specific Integrated Circuit
FDB	Forwarding Database
FIB	Forwarding Information Base
IP	Internet Protocol
LPM	Longest-Prefix Match
MAC	Media Access Control
MDB	Multicast Group Database
RIF	Router Interface
SDK	Software Development Kit
TCAM	Ternary Content-Addressable Memory
UFT	Unified Forwarding Table
VRF	Virtual Routing and Forwarding

References

[1] Charles E. Spurgeon and Joann Zimmerman. "Ethernet Switches". First Edition. April 2013. ISBN: 978-1-449-36730-5. O'Reilly books.

[2] Huang, Paul. "Switch Concept & Architecture." Switch Router Design & Implementation course. National Chiao-Tung University, speed.cis.nctu.edu.tw/~ydlin/course/cn/srouter/Lesson3.pdf. Retrieved 2018-02-24.

[3] Mellanox Technologies. "SN2700 Open Ethernet Switch". Product Brief. http://www.mellanox.com/related-docs/prod_eth_switches/PB_SN2700.pdf. Retrieved 2018-02-24.

[4] Cisco Public "Cisco Nexus 9200 Platform Switches Architecture White Paper". September 2016. https://www.cisco.com/c/dam/en/us/products/collateral/switches/nexus-9000-series-switches/white-paper-c11-737204.pdf. Retrieved 2018-02-25.

[5] Arista Networks. Arista 7500 Switch Architecture ('A day in the life of a packet'). https://www.arista.com/assets/data/pdf/Whitepapers/Arista_7500E_Switch_Architecture.pdf. December 2016. Retrieved 2018-02-25.

[6] Linux Kernel Documentation. "Universal TUN/TAP device driver" https://www.kernel.org/doc/Documentation/networking/tuntap.txt. Last Revision: 2002. Retrieved 2018-02-26.

[7] Linux Kernel Documentation. "Ethernet switch device driver model (switchdev)". Last Revision: September 2017. https://www.kernel.org/doc/Documentation/networking/switchdev.txt. Retrieved 2018-02-27.

[8] Scott Feldman. "Rocker: switchdev prototyping vehicle". Proceedings of the Netdev 0.1 Conference, February 2015, Ottawa, On, Canada.

[9] Advanced Configuration and Power Interface Specification, Version 6.2 Errata A" (PDF). UEFI.org/specifications. September 2017. Retrieved 2018-02-21.

[10] David Ahern. "net: VRF Support". LWN Article. February 2015. https://lwn.net/Articles/632522/. Retrieved 2018-03-02.

[11] Roopa Prabhu, Wilson Kok. "Hardware accelerating Linux network functions". Netdev 0.1. February 2015.

[12] Toshiaki Makita. "Hardware Accelerating Linux Network Functions Part I: Virtual Switching Technologies in Linux". Netdev 0.1. February 2015.

[13] Mellanox Technologies, Inc. "Mellanox Switchdev Ethernet, Release Notes". http://www.mellanox.com/related-docs/prod_software/Switchdev_release_notes_v1.0.pdf. April 2017. Retrieved 2018-02-21.

[14] W3Techs. "Usage Statistics and Market Share of Unix for Websites". https://w3techs.com/technologies/details/os-unix/all/all. March 2018. Retrieved 2018-02-10.

[15] Ars Technica. "Linux is king *nix of the data center - but Unix may live on forever". https://arstechnica.com/information-technology/2013/10/

linux-is-king-nix-of-the-data-center-but-unix-may-live-on-forever/. October 2013. Retrieved 2018-02-10.

[16] Cisco Systems. "Cisco NX-OS Software: Business-Critical Cross-Platform Data Center OS". https://www.cisco.com/c/en/us/products/collateral/switches/nexus-7000-series-switches/white_paper_c11-622511.html. August 2014. Retrieved 2018-02-10.

[17] Telecomlead. "Switch and router market share of Cisco, Huawei and HPE". http://www.telecomlead.com/telecom-equipment/switch-router-market-share-cisco-huawei-hpe-79457. September 2017. Retrieved 2018-02-10.

[18] Jiri Pirko, Ido Schimmel, Matty Kadosh. 2016. Switchdev BoF, Netdev 1.2. https://netdevconf.org/1.2/slides/oct5/08_switchdev-BOF.pdf. Retrieved 2018-02-10.

[19] Arkadi Sharshevsky. "Driver profiles RFC" https://www.mail-archive.com/netdev@vger.kernel.org/msg181492.html. Retrieved 2018-02-10.

[20] Arkadi Sharshevsky. "ASIC Pipeline Debug API". https://netdevconf.org/2.1/papers/dpipe_netdev_2_1.odt. Retrieved 2018-02-10.

[21] "Changes to spectrum driver to offload FIB entries inline with user request". https://github.com/CumulusNetworks/net-next/tree/res-mgmt/sync-ipv4-fib. Retrieved 2018-02-10.

[22] "Changes to spectrum driver to simulate the credit-based solution for inline synchronous resource checks". https://github.com/CumulusNetworks/net-next/tree/res-mgmt/credit-based-fib. Retrieved 2018-02-10.

[23] "Changes to spectrum driver to simulate offload failure resulting in blackholed packets". https://github.com/CumulusNetworks/net-next/tree/res-mgmt/fib-offload-blackhole. Retrieved 2018-02-10.

[24] Andy Roulin, Shrijeet Mukherjee, David Ahern, Roopa Prabhu. "Resource Management for Hardware Accelerated Linux Kernel Network Functions". Netdev 2.2, Seoul, Korea. November 2017. https://www.netdevconf.org/2.2/papers/roulin-hardwareresourcesmgmt-talk.pdf. Retrieved 2018-02-10.

[25] Quagga Routing Software Suite. https://www.nongnu.org/quagga/. Retrieved 2018-02-10.

[26] FRRouting Routing Software Suite. https://frrouting.org/. Retrieved 2018-02-10.

[27] Postel, J. (1981). Internet Protocol. RFC 791 (Internet Standard). RFC. Updated by RFCs 1349, 2474, 6864. Fremont, CA, USA: RFC Editor. doi: 10.17487/RFC0791. url: https://www.rfc-editor.org/rfc/rfc791.txt.

[28] Deering, S. and R. Hinden (2017). Internet Protocol, Version 6 (IPv6) Specification. RFC 8200 (Internet Standard). RFC. Fremont, CA, USA: RFC Editor. doi: 10.17487/ RFC8200. url: https://www.rfc-editor.org/rfc/rfc8200.txt.

[29] Cumulus Linux. https://cumulusnetworks.com/products/cumulus-linux/. Retrieved 2018-02-10.

[30] "IEEE Standard for Ethernet" (2016). In: IEEE Std 802.3-2015 (Revision of IEEE Std 802.3-2012), pp. 1–4017. doi: 10.1109/IEEESTD.2016.7428776.

[31] "IEEE Standard for Local and metropolitan area networks: Media Access Control (MAC) Bridges" (2004). In: IEEE Std 802.1D-2004 (Revision of IEEE Std 802.1D-1998), pp. 1–277. doi: 10.1109/IEEESTD.2004.94569.

[32] "IEEE Standard for Local and metropolitan area networks–Bridges and Bridged Networks" (2014). In: IEEE Std 802.1Q-2014 (Revision of IEEE Std 802.1Q-2011), pp. 1– 1832. doi: 10.1109/IEEESTD.2014.6991462.

[33] Postel, J. and J.K. Reynolds (1988). Standard for the transmission of IP datagrams over IEEE 802 networks. RFC 1042 (Internet Standard). RFC. Fremont, CA, USA: RFC Editor. doi: 10.17487/RFC1042. url: https://www.rfc-editor.org/rfc/rfc1042.txt.

[34] Red Hat. "8.10. NIC OFFLOADS". https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/network-nic-offloads. Retrieved 2018-02-10.

[35] Mellanox Technologies, Inc. "Mellanox ASAP^2: Accelerated Switching and Packet Processing" http://www.mellanox.com/related-docs/products/SB_asap2.pdf. Retrieved 2018-02-10.

[36] Dave Watson. Facebook. "KTLS: Linux Kernel Transport Layer Security". Netdev 1.2. https://netdevconf.org/1.2/papers/ktls.pdf. Retrieved 2018-02-10.