Hybrid Switch: Dynamic Flow Rule Offloading on High Performance Networking Hardware

Master-Thesis

Marvin Härdtlein
KOM-M-0715

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Elektrotechnik
und Informationstechnik
Fachbereich Informatik (Zweitmitglied)

Fachgebiet Multimedia Kommunikation
Prof. Dr.-Ing. Ralf Steinmetz

Hybrid Switch: Dynamic Flow Rule Offloading on High Performance Networking Hardware
Dynamische Verteilung von Flussregeln zwischen Software und Hardware Netzwerk-Switches

Master-Thesis
Studiengang: Informatik
KOM-M-0715

Eingereicht von Marvin Härdtlein
Tag der Einreichung: 18. Dezember 2020

Gutachter: Prof. Dr.-Ing. Ralf Steinmetz
Betreuer: Ralf Kundel, Christoph Gärtner

Technische Universität Darmstadt
Fachbereich Elektrotechnik und Informationstechnik
Fachbereich Informatik (Zweitmitglied)

Fachgebiet Multimedia Kommunikation (KOM)
Prof. Dr.-Ing. Ralf Steinmetz

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Marvin Härdtlein, die vorliegende Master-Thesis gemäß § 22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, den 18. Dezember 2020

_____

Marvin Härdtlein

Contents

1 Introduction **3**
- 1.1 Motivation & Contribution . . . 4
- 1.2 Outline . . . 4

2 Background **5**
- 2.1 The Internet . . . 5
  - 2.1.1 IPv4 & IPv6 . . . 6
  - 2.1.2 Border Gateway Protocol . . . 7
  - 2.1.3 Longest Prefix Match . . . 7
- 2.2 Software-defined Networking . . . 7
  - 2.2.1 OpenFlow . . . 9
  - 2.2.2 Proactive vs Reactive Flow Setup . . . 9
- 2.3 Network Function Virtualization . . . 10
  - 2.3.1 DPDK . . . 11
  - 2.3.2 SmartNICs . . . 11
- 2.4 P4 Language . . . 12
  - 2.4.1 P4_14 . . . 13
  - 2.4.2 P4_16 . . . 13
  - 2.4.3 Compiler . . . 13
  - 2.4.4 Barefoot Tofino . . . 14
- 2.5 Summary . . . 14

3 Related Work **15**
- 3.1 Offloading Feasibility . . . 15
  - 3.1.1 Traffic-aware Flow Offloading . . . 16
- 3.2 Inter-Flow-Rule Dependencies . . . 17
  - 3.2.1 The Problem . . . 17
  - 3.2.2 Solution I: FIB Caching . . . 18
  - 3.2.3 Solution II: CacheFlow . . . 21
- 3.3 Other Use cases for Hybrid Switches . . . 24
  - 3.3.1 ShadowSwitch . . . 24
  - 3.3.2 OpenState . . . 26
- 3.4 Summary & Analysis . . . 28

4 Design **29**
- 4.1 High Level Architecture . . . 30
  - 4.1.1 Data Plane Splitting . . . 30
  - 4.1.2 Control Plane . . . 31
  - 4.1.3 Sum Up . . . 32
- 4.2 Data Plane . . . 32
  - 4.2.1 L3 Routing Pipeline . . . 32
  - 4.2.2 L2 Compliance . . . 35
  - 4.2.3 Metrics . . . 39
  - 4.2.4 Splitted Dataplane with common definition language . . . 39

4.3 Control Plane . . . 42
4.3.1 Interfaces . . . 43
4.3.2 Routing Information Base . . . 45
4.3.3 Metric collector . . . 48
4.3.4 Offloading . . . 49
4.4 Summary . . . 52

5 Implementation **53**
5.1 P4HC Data Plane . . . 53
5.1.1 P4HC Cache . . . 53
5.1.2 P4HC Forwarder . . . 57
5.2 Metric Collector . . . 61
5.3 P4HC Control Plane . . . 62
5.3.1 North- and southbound interfaces . . . 63
5.3.2 Routing Information Base . . . 66
5.3.3 Configuration . . . 68
5.3.4 Startup . . . 69
5.3.5 Offloading . . . 70
5.4 Summary . . . 70

6 Evaluation **73**
6.1 Evaluation Setup . . . 73
6.2 P4HC Cache Evaluation . . . 74
6.2.1 Tofino Hardware Utilization . . . 75
6.2.2 P4HC Cache Throughput . . . 75
6.3 P4HC DPDK-Forwarder Evaluation . . . 78
6.4 P4HC Offloading Evaluation . . . 82
6.4.1 Normal Operation . . . 83
6.4.2 Overloaded Operation . . . 84
6.4.3 Detailed measurements . . . 86
6.5 P4HC Control Path Evaluation . . . 88
6.5.1 P4HC DPDK-Forwarder . . . 89
6.5.2 P4HC Cache . . . 90
6.6 Functional P4HC Evaluation . . . 93
6.7 Summary . . . 93

7 Conclusions **95**
7.1 Contribution & Results . . . 95
7.2 Future Work . . . 96

Bibliography **97**

Abstract

A hybrid switch is the approach of combining a programmable switch with an off-the-shelf x86 server. The goal is to replace purpose built networking devices like routers or to solve the issue of lacking table-space on SDN enabled switches. However, most modern programmable switches have the capability to perform actions on multiple layers of the OSI reference model, mostly up to layer 4. This allows them to perform tasks like routing or flow-based forwarding with protocols like OpenFlow. The main issues are that, compared to the purpose built hardware, programmable switches only have a small amount of memory to store the forwarding rules. To compensate the lack of table-space in those switches, a software switch on a standard x86 server can be used. The software switch has mostly enough memory to store all available forwarding rules. Afterwards, only rules responsible for high traffic volumes are installed into the hardware switch and the remaining traffic is forwarded at the server. A new rising challenge in those hybrid switches are complex dependencies between flow rules, which could create an incorrect forwarding behavior if not treated sufficient. Also, there are additional requirements to the data plane like egress management and sufficient forwarding performance at the x86 software switch.

In this thesis the related work is examined for existing approaches of hybrid switches, with the result that the most research was performed at the controller side of hybrid switches. Astonishingly there was almost no work at the data plane side. So, this work will contribute a new approach for a high performance, programmable and protocol independent data plane for hybrid switches called P4HC P4 Hybrid Cache. Based on state-of-the-art technologies like the P4 language and DPDK, a prototype, for the use case of IPv4 full table routing, is implemented and contributed as a new platform for further research on hybrid switch data planes.

The presented and implemented design for P4HC consist out of two data planes. The first data plane is called forwarder, which contains all available rules within P4HC. In the prototype the forwarder was implemented based on the DPDK framework. The second data plane is called cache, which is responsible for the most active rules. The cache was implemented as a P4 program which was executed on a Barefoot Tofino hardware switch. Also, a controller was implemented to coordinate the two data planes and to exchange routes with neighbour routers.

Finally the prototype was evaluated with the following results: The Tofino cache data plane could store 110.592 route entries and was able to handle traffic at 100 Gbit/s of line rate at an average latency of around 660 nanoseconds. The forwarder was able to achieve 40 Gbit/s line rate on larger packet sizes and 17 Gbit/s at 21,9 Mpps on smaller packet sizes. The average introduced latency through the forwarder on optimal conditions is around 8 microseconds. When traffic is moved from the forwarder to the cache, to reduce the load of the forwarder, no unexpected behavior could be observed in normal non-overloading situations. Finally, the rate on which P4HC can update its routes in the data planes were evaluated.

The conclusion, of the work is that the P4HC prototype performs better than expected. Also, P4HC proves the first time that decent performance in hybrid switches is possible while maintaining full programmability and protocol independence. Also, P4HC proves that a performant hybrid switch data plane can compete with enterprise grade routes in certain scenarios.

1 Introduction

Hybrid switches are a new approach of combining the flexibility of programmable switches with the power of packet forwarding on standardized hardware. Those hybrid switches promise to solve the issues of the lack of table space on programmable network switches. Programmable switches are required for modern network technologies like software-defined-networking (SDN).

SDN is the approach of separating the control from the data plane, which allows to fully program a network from a logical centralized controller [KRV+15]. This concept rethinks the conventional design of networks where for each task a specialized proprietary network device is used. However, SDN enabled networks also need networking hardware to process the packets in the data plane. Mostly, programmable switches are used, because they are relatively cheap and have a lot of forwarding capacity. They also come with the capability of interacting with packets on upper layers in the OSI model [Zim80]. Legacy switches only work on OSI layer 2 whereas programmable switches can usually operate up to layer 4 or even higher. Those switches enable the substitution of application-specific devices like routers. Routers are typically very capable and thus expensive network devices, which are responsible for routing packets to their destination network within the Internet. To find the right destination for each packet, they use their routing table, which is usually a so called full table. The full table is a collection of all routes that are required to reach any destination network within the Internet. Since the Internet is growing rapidly, the full table is growing as well. Up today the full table has reached nearly 850.000 routes [cid] that are required in order to reach all destination networks. For the typical routers this is not an issue, since they are purpose-built for this use case and have enough memory to hold millions of routes.

However, even if the programmable switch can perform routing on layer 3, it still has the major issue that there is not enough table-space (around 100.000 routes, see Chapter 6) in the switches hardware to store a full table. Therefore, if a programmable switch should substitute a router, it must be able to handle the full routing table. This is the point where hybrid switches are coming in. The programmable switch can store only a certain number of routes in its hardware. For this limited set of routes, the programmable switch can outperform legacy routers in terms of performance per price and density. In order to also be able to handle a full table, the switch needs some support to handle all routes.

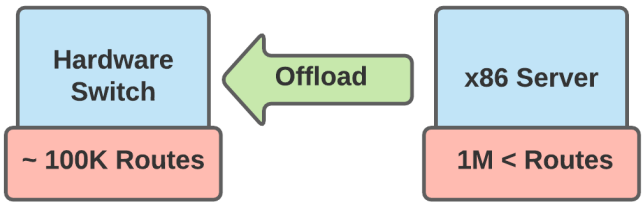


Figure 1.1: Shows how a x86 server is used to store all routes and offloads the most active ones to a hardware switch.

Standard of the shelf servers based on common x86 technology are typically optimized for computing, so they can interact with large amounts of memory. This allows them to store millions of routes easily. Initially, standard servers were not designed to forward packets at high rates and neither were the operating systems designed for packet forwarding. However, there are new software frameworks that allow to decouple the packet processing from the operating system to achieve more performance. Frameworks like the Dataplane Development Kit (DPDK) [dpdb] enable the forwarding of line-rate of 100 Gbit/s [pena, mel] and more on common x86 hardware. While these numbers are impressive, they are still relatively slow compared to hardware-based switches and routers, which can achieve rates beyond 35

Tbit/s [mxj]. The main advantage is however the previously mentioned memory capacity in order to store almost any amount of routes. So, both the software forwarding and the hardware switch have their weaknesses that could be compensated by the other. If both are combined, they can hold plenty of routes and achieve a comparable performance to hardware routers. This can be implemented by installing all available routes into the software and then only install actively used or more heavily used routes into the switching-hardware. This principle is also visually provided in Figure 1.1. This hybrid concept can be applied to any kind of application or use-case where programmable switches reach their memory limits, like in SDN environments for example.

1.1 Motivation & Contribution

To evaluate if this technology is a promising approach, this work will investigate existing implementations of hybrid switches, with the following lead questions:

- Which approaches already exist for hybrid switches and how do they work?
- What are the performance characteristics of hybrid switches?

To answer those questions, the literature will be examined to show existing work. The research will show that there are many works that have mainly contributed to the controller logic in hybrid switches. It will reveal a lack of work on the data plane on hybrid switches. So this work will contribute a new approach for a hybrid switch, which is focused on the maximum achievable performance, programmability and flexibility on the data plane. A prototype will be developed, based on the example of a full table Internet router in order to prove that with the right data plane it's possible to compete with hardware routers by using standard of the shelf switch and server hardware.

1.2 Outline

This work is structured as following: In chapter 2 an introduction to related topics is presented that are key to understand the basics of this work. In chapter 3 the related work is presented and investigated. Afterwards, chapter 4 presents a new approach for a data plane centric hybrid switch called P4HC. Also, the design of P4HC will be presented and discussed. Then in chapter 5 the implementation details of P4HC are presented. In Chapter 6 an evaluation of P4HC is provided and examined. Finally, in the last Chapter 7 a conclusion and outlook is provided.

2 Background

In the following chapter an introduction to relevant topics is given. First, the high level structure of the Internet will be explained. Afterwards, key technologies like software-defined-networking, network function virtualization and the P4 language are introduced.

2.1 The Internet

The Internet allows end-to-end communication of various devices around the globe. Completely drilled down, the Internet consists of many individual networks. Those networks are connected to each other and thus enable end-to-end communication between two devices residing within in those networks. Each of this individual networks is called an autonomous system (AS)[HB96] and receives an autonomous system number (ASN). The ASN is globally unique and is assigned by a regional Internet registry.

Figure 2.1 presents an exemplary overview of the Internet. On the right side of the image, a client computer is located and on the left side a content provider is located. Now if the client wants to request content like a video stream from a content provider, the data must be transmitted between the client and the provider. As the figure shows, there are many different networks in the path between the client and the content provider. They will all transport data, but with different characteristics. The most important type of networks are Internet Service Provider (ISP), Carrier, Internet Exchange Points (IXP) and Content Provider Networks.

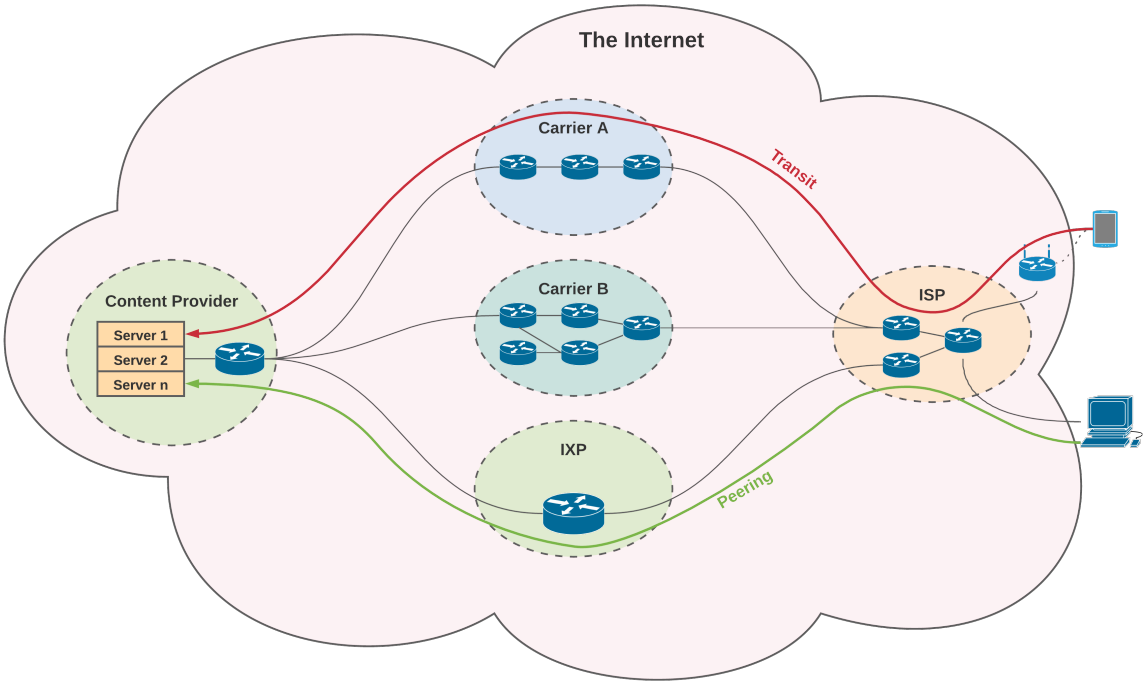


Figure 2.1: Shows how the Internet consist out of many different interconnected networks to deliver data from end to end.

An ISP network connects end users to the Internet [Hec07]. The connection can be established over different access technologies like DSL, via coaxial cable, via fibre or radio for mobile devices [Bin06].

After the ISP has enabled the communication with the end device, it will send the user's data to an upstream carrier or directly to a content provider network via a peering link, if available. Carriers are essentially the backbone of the Internet. They operate huge networks that can span around the globe. Their networks enable the communication between any network within the Internet. The service to use their networks is called IP-Transit and must be paid based on usage. In our example the client wants to request content from a content provider. Since the ISP network cannot directly access the content providers network the data need to be transmitted using the carrier network. If many clients within an ISP network request content from the same content provider, it quickly can happen that the IP-Transit becomes too costly to be economical for the ISP. If two networks exchange a lot of data between each other, they are often trying to connect directly. This allows them to avoid using costly carrier networks as much as possible. This approach is called peering. Since direct interconnects can be costly, there exist places called Internet Exchange Points (IXPs) [CSF13]. They enable direct peering between two or more networks for a fixed price. This allows the networks to bypass the carrier networks to reduce cost. Also, the quality of the connection can be increased on an IXP, since the path between the networks becomes shorter. In the end, there are the content provider networks. These networks are essentially the end point and are often called stub networks, because they do not enable communication to other networks. Within these networks many of servers are located that provide the requested content to the user.

In reality the classification of a network can become more complicated. There are many companies out there that operate networks which provide multiple service. For example, the Deutsche Telekom AG operates a carrier network, a content provider network as well as an ISP network.

The Figure 2.1 also shows up that there are mostly multiple paths between two networks. So, there are multiple ways that a network can reach another one. To answer the question which path to take, routing algorithms are used. Routing algorithms enable routers to exchange information about their available paths to their neighbour routers. Based on the paths learned from the neighbour routers, each router can build a local routing table to decide how to deliver packets. There are two main classes of routing protocols for different use-cases. First there are the interior gateway protocols (IGP) which are used within a network and on the other hand, there are exterior gateway protocols (EGP) which are used between networks/autonomous systems. One commonly used EGP protocol is the Border Gateway Protocol (BGP) which will be explained in the next section.

2.1.1 IPv4 & IPv6

Before BGP can be explained in detail, it is important to understand how addressing is realized within a network. Addressing is required to identify each host in a network uniquely. There are two different addressing schemes today.

The former standard is the Internet Protocol Version 4 (IPv4)[rfc81]. IPv4 builds addresses out of 32-bits. These addresses can be represented as four decimals like 1.1.1.1, called the dotted-decimal notation. With the addition of a subnet mask, it is possible to group IP addresses into smaller networks. To address those smaller networks, a 32-bit mask is used to build up an address for the smaller networks. The subnet mask identifies which bits from the IP address are used to identify the network. The address that is created by applying the bitmask to an IP address is then called network address. A network address combined with its corresponding subnet mask is the called prefix. Those prefixes are also used to simplify the routing process. Instead of exchanging the information where each IP address is located, routers will only exchange information about where a prefix is located and thus essentially group many addresses into one route.

Since the Internet was growing rapidly, IPv4 cannot serve the increasing demand of addresses, so the Internet Protocol Version 6 (IPv6) [DH17] was introduced which uses 128-bit wide addresses.

2.1.2 Border Gateway Protocol

The Border Gateway Protocol (BGP) [RLH+94] is used to exchange routing information between two autonomous systems. In order to exchange routes, both routers must set up a BGP Session over which routes are advertised and received. A route in BGP consist of an IP prefix and an AS path. The IP prefix enables BGP to implement Classless Inter-Domain Routing (CIDR) [FLYV93]. The AS path is used as distance metric. Per default BGP selects the best path based on the length of the AS path. If there are two routes from two peers (neighbor routers), the router will select the route with the shortest path length. In Figure 2.2 is shown how the path is built up trough the exchange of routing information between routers. Router A with AS1 originates the prefix 1.1.1.0/24. Then it announces the route to router B with its own ASN appended to the AS path. Router B receives the prefix with an AS path length of one. When router B announces the route again to router C, it will also append its AS number to the path, so router C receives the route with the AS Path AS2 AS1. Finally, BGP builds up a local routing table based on the best paths it has learned from its neighbors.

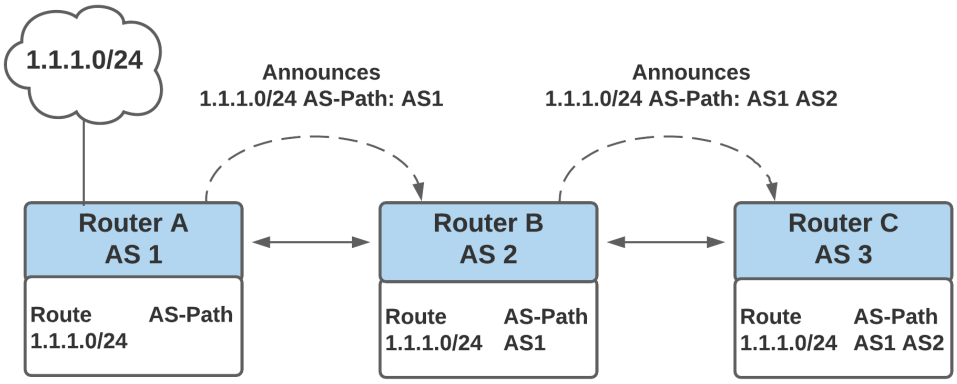


Figure 2.2: Depicts how BGP is exchanging route information and builds up the AS path.

2.1.3 Longest Prefix Match

In the previous section BGP was introduced and an overview about how BGP is used to build up a local routing tables was given. To use a routing table for making a packet forwarding decision, a longest prefix match (LPM) algorithm is required. The longest prefix match is used to identify the most specific prefix for a given destination address in a routing table. The most specific prefix is the prefix with the longest subnet mask that still matches the destination address. An example routing table is given in Figure 2.3. Now if a packet with destination 1.1.1.1 is received at a router with the given routing table, there are two prefixes which contain 1.1.1.1: 1.1.1.0/24 and 1.1.0.0/16. The longest prefix match will select the route 1.1.1.0/24, because the subnet mask of /24 is longer than the subnet mask of /16, so it is more specific. Afterwards the packet is sent to the corospondig next hop X.

2.2 Software-defined Networking

Software Defined Networking (SDN) is the new concept of decoupling the control plane from the data plane by moving the control plane to a logical centralized point within the network [KRV+15]. In comparison to conventional networking where each box has its own control plane tightly coupled to the data plane, SDN allows to program the network from a single point of view. This allows to remove the complexity of distributed protocols from each device. This paradigm shift can also be seen as a transition from

Prefix	Next-Hop
212.224.0.0/18	B
1.1.1.0/24	X
1.1.0.0/16	Y
192.168.1.0/24	Z

Figure 2.3: Shows an example routing table which is used with a longest prefix match.

a horizontal to a vertical control plane communication. This shift is visualized in Figure 2.4. Common devices each have their own control plane which communicates with its neighbors. The control plane in routers for example requires each router to run BGP on its own, in order to exchange route information with its neighbors. Afterwards, each individual router computes the optimal paths based on its own view of the network. In the new software-defined approach, there is no communication on the control plane towards other individual devices anymore. Instead a new well-defined API between the device and the control plane (also called controller in SDN terms) is introduced. This allows to move the control plane up to be independent of the underlying hardware and operate in a virtual logically centralized fashion. Instead of running a routing daemon on each router, we now have the centralized controller with its global network view, that can compute all paths within the network and install the them centralized into the individual boxes' data planes. This removes the need of complex, vendor-specific and often even error prone algorithms on many devices.

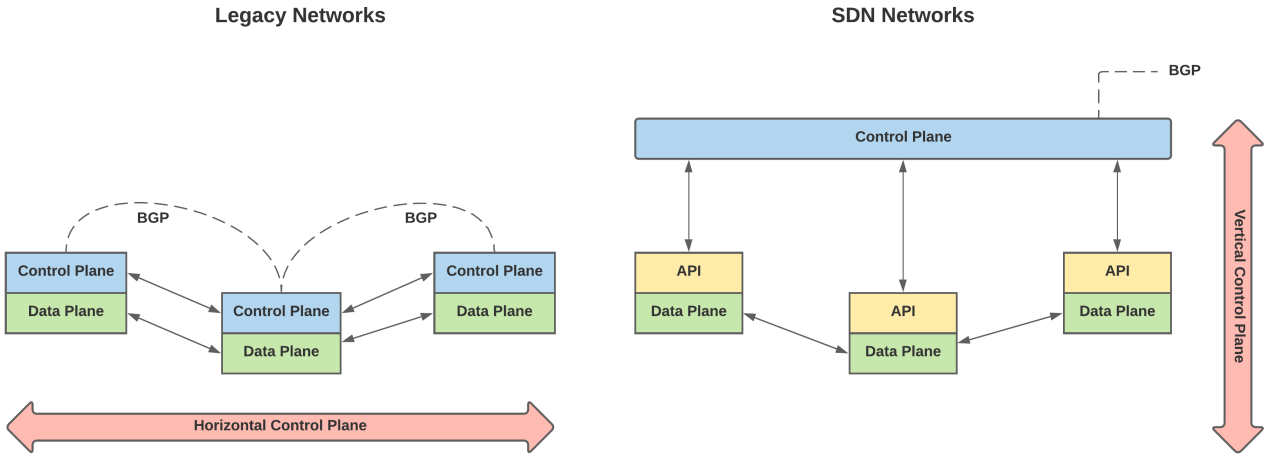


Figure 2.4: Shows the paradigm shift from horizontal to vertical control plane interaction. Inspired By [Ozd12].

SDN goes even further: now the network becomes completely programmable for the controller, so it can be integrated into applications that run on top of the network. An example would be a large-scale virtualization environment where many virtual machines (VM) are connected to a single network. When a VM is migrated between to servers, the path from and to the new location must be changed across the whole network in order to avoid any outage during migration. Traditionally, the MAC learning happens after the migration of the VM which could cause a short connectivity outage. With SDN, a Virtualization Controller which initiates the migration now can signal the network the new location of the VM and thus prevent any impact due to MAC learning delay. The network can now be coordinated to setup the new path to eliminate the time-consuming learning mechanisms and deliver migrations without any impact. Another use case would be network virtualization [BBRK16] where a common large network is sliced

down into multiple smaller networks. These smaller networks are logically independent, but share the same hardware resources. The principle of network slicing also exits in 5G Networks [FPEM17, ARS16].

However, these concepts require a well-defined API between the controller and the data plane. From the controller point of view this interface is called the Southbound Interface. The most popular example for a protocol to control the data plane is OpenFlow, which is briefly introduced in the next section.

2.2.1 OpenFlow

As mentioned previously, SDN requires a well-defined API between the controller and the data plane. The most popular standardized protocol is OpenFlow [MAB+08]. It has become the defacto standard for SDN nowadays. When a network device is in an OpenFlow mode, it has no own control plane logic anymore and provides a well-defined, reliable, and secure interface to externally program its forwarding behavior.

OpenFlow allows to program the network device in a flow-based fashion. To model this behavior, OpenFlow introduces flow rules which allow to match a packet to a flow and then perform actions on the matching packets. OpenFlow allows to identify a flow based on the packet header from OSI layer 2 up to layer 4 and includes additional metadata like the physical ingress port on the hardware. To match against these attributes, OpenFlow supports exact matches as well as wildcard matches. The wildcard match allows to make use of a do-not-care bit, which enables matching of subnet prefixes, for example. After a successful match, each flow rule has a set of actions that allow to process the packets. Actions can be used to forward the packet to an egress port, modify the packet, drop it, or send it up to the controller for further processing.

Mainly, the flow rules consist of the match and action part, but there are some more elements of a flow rule that are worth to be mentioned. There is a counter which allows to monitor how many packets have been matched against a flow rule, a priority which allows to weigh flow rules against each other and a timeout which allows to remove the flow rule after a given period of time from the table. Last but not least, there's an idle timeout which enables to remove a rule, if no packet is matched against it within a specified time period and thus allows the rule to automatically age out.

All flow rules are stored in a flow table which is programmed to the forwarding hardware. Depending on the capabilities of the device and the OpenFlow version, multiple flow tables are possible to be created and chained up. Since OpenFlow was designed in mind to reuse existing hardware and enable widespread compatibility, it misses some flexibility. There is for example the issue that the capabilities are limited to the features of the protocol itself or the limitation of the underlying hardware. For the end user there are no options to customize it afterwards, because the hardware is fixed and OpenFlow does not describe a standard to define certain behaviors. Even if the hardware would support the features, a new revision of the OpenFlow protocol would be required to expose the compatibility to the controller.

2.2.2 Proactive vs Reactive Flow Setup

In SDN networks there are two ways of setting up flow rules. The first one is the proactive setup which means that the rules are installed proactively by the controller into the hardware, waiting for a packet to match. This approach removes any flow setup delays, but on the other hand also wastes valuable space in the limited flow table, if a rule is not used at the time. This approach of flow installation is often appropriate in more static networks where changes do not occur that often.

The alternative is the reactive flow rule setup. Here flow rules are installed only if they are required. Triggered by an event like a Packet_In message of a switch, the controller computes the required flow rules for handling the packet and then installs the new flow rule(s) into the switch. If there are no more packets matching this rule, the rule will be removed after a given timeout to free up space in the flow table. This approach reduces the flow table size, but also introduces some latency on the first packets of a flow due to the setup delay of the flow rules in the data plane.

Finally, hybrid approaches are also possible. Rules that are used more often will be installed proactively and the rest is handled reactively. A mixed setup can also be used to enable further optimizations like prediction or signaling. An example would be a backup job initiated from a backup controller. The job runs every night at the same time. Normally the rules need to be installed at all times or must be installed reactively on a Packet_In event. Since the job runs every night, the controller could learn that every night the ruleset x must be installed in order to run the job successfully. Then the controller could install the rules proactively based on his prediction that the job will run and reduce the setup time. In the signaling case, the backup controller would signal the SDN controller that it wants to run a backup job on the network. The controller then installs the required rules into the network, so that when the backup job starts no more flow setup is required.

This section shows that there are many ways to setup a flow rule. The Important part is to understand the main concepts of reactive and proactive installation of rules. Those are the primitives on which other techniques will build up from.

2.3 Network Function Virtualization

In today's networks it's difficult to deploy new network services, due to the increasing complexity of modern networks. Network services like firewalls or Deep Packet Inspection (DPI) devices are typically provided by proprietary hardware devices. To deploy a new service, mostly a new device must be ordered and shipped to a data-center before the integration into the network can be performed. This process consumes time, long term planning and reduces the ability to deploy and scale services on demand. To address this problem NFV or Network Function Virtualization [LC15] was introduced. A Virtual Network Function (VNF) could be a firewall for example. The Firewall or the VNF can then be executed on top of a virtualized environment on common x86 hardware. This approach enables the deployment of network functions on mostly any modern x86 platform which enables this kind of virtualization technology. The x86 platform enabling the virtualization is also referred as hyper-visor. On top of the hyper-visor multiple network functions can be executed in sequence or in parallel [ZLZ+16], like for example the firewall. The gained flexibility through virtualization enables quick deployment and scaling of new network functions on demand. To further reduce the need of proprietary hardware, the hyper-visors are based on off-the-shelf hardware. This allows to use any common x86 server from any vendor for executing the network functions. As promising the approach is on the first glance, there are also new problems that need to be solved to enable a true NFV enabled network environment approach.

To enable a homogeneous platform that can execute Virtual Network Functions, an open architecture is required. One platform design is the ETSI standard [ETS13], which provides an open platform for a Network Functions Virtualization Infrastructure (NFVI). It provides everything that is required for managing and executing VNFs. The important domains are the compute domain, which provides the virtualization environment wherein the functions are executed, the Infrastructure Network Domain which enables connectivity to and between the VNFs and at least the Management and Orchestration Domain which deploys the VNFs and coordinates the infrastructure. However, there are also many other challenges in NFV besides the infrastructure alone. One issue is the connection of the real network and the VNFs. Here SDN comes in handy. SDN allows to adapt the network quickly and move traffic from the previous path to the newly deployed VNFs. Also, in the NFVI SDN comes into play. Here SDN is used to set up paths between two VNFs, when two VNFs need to be executed in series. The connection of multiple VNFs is called VNF chaining. Another big issue in NFV is the placement problem. At which place within the network should the VNF be deployed? There are many parameters that influence the selection such as latency, bandwidth, costs of transport and cost of starting a new instance [MSG+16].

But even if all challenges can be solved sufficiently, there is still the issue with the performance of the VNF. Standard network devices like enterprise grade firewalls or routers are developed to serve the network at line rate. To reach the required speed they use specialized hardware, which is only designed for one specific use-case. Since VNFs are running within standard virtual machines - mostly

based on a Linux operating system - they cannot match the performance of application-specific hardware devices. To overcome those performance drawbacks, user-space networking is required to enable faster packet processing. This will not enable a VNF to compete with purpose-designed networking hardware, but enables them to deliver a kind of compatible good-enough performance, with the main advantage of flexibility. One framework which enables user space packet processing is DPDK, which will be introduced in the following.

2.3.1 DPDK

The Data Plane Development Kit (DPDK) [dpdb] is a software framework which enables user-space packet processing on common x86 hardware with Linux as operating system. To enable user space network access, DPDK moves the Network Interface Card (NIC) from the operating systems kernel-space to the user-space. Normally network communication on Linux based systems needs to pass through the kernel's networking stack. When a packet arrives at the NIC, the packet is copied from the NIC to the kernels memory area and an interrupt is generated. The kernel then handles the interrupt and copies the packet again to the appropriate process's user-space memory area. When the process sends a reply, the packet traverses the same path in reverse order again. The multiple copying of the packets in memory and the handling of interrupts consumes many additional computational resources. This overhead prevents the normal network stack of Linux to reach a line-rate like performance.

As mentioned before DPDK solves this by directly accessing the NIC from the application running in user-space. This removes the overhead of copying the packet by the kernel. Also, DPDK uses a poll mode driver instead of an interrupt-based driver. Instead of reacting to new packets, the poll mode driver continuously checks if new packets have arrived and then processes them. This polling requires more CPU time than interrupts at first, but has the advantage that the processing of packets cannot be interrupted and thus avoid performance penalty due to context switches. For example, a 10 Gbit/s interface can handle around 14 million packets per second, which would lead to roughly the same number of interrupts. If interrupts would be used to handle that number of packets per second, the CPU would not have any CPU time left to process any packets after handling the interrupts alone. DPDK even uses a more efficient approach on polling. Instead of polling each packet individually, DPDK polls the packets in small batches. This reduces the effort of polling every single packet and additional CPU time is preserved, which can in turn be used to process the packets. DPDK also provides many other optimizations to achieve more performance compared to kernel networking.

As a result of all these optimizations DPDK can achieve line-rate performance, even on 100 Gbit/s [pena, mel].

2.3.2 SmartNICs

While frameworks like DPDK can be flexible and powerful for packet processing, there are many situations where some part of the application could be offloaded to a NIC. Common NICs enable offloading functions to support the application like vlan offloading and TCP/UDP/IP checksum computations. When vlan offloading is used the NIC will parse the vlan header information for the application, which will save additional CPU cycles. A more advanced offloading approach is the use of SmartNICs [LCS+19]. SmartNICs enable programmability at the NIC level. Those NICs have specialized hardware that enables to perform operations on packets at the NIC-level. This specialized hardware is able to handle certain operations and workloads much faster and more efficient than the application on the server could. Therefore, the hardware can decrease the load on the application, which improves the overall performance of the system even further. Use cases could be flow-table offloading or crypto offloading for IPsec, for example.

There are also SmartNICs by Netronome [net] and Pensando [penb] where the complete packet processing pipeline can be defined in the P4 language.

2.4 P4 Language

In the previous section we have seen in many ways that in modern networking flexible and programmable hardware is required. Protocols like OpenFlow were designed to be compatible with existing switching devices to enable wider adoption. The limitation from this approach is that the switching hardware is fixed and no new functionality can be added later on. To enable real programmable hardware the P4 language was introduced.

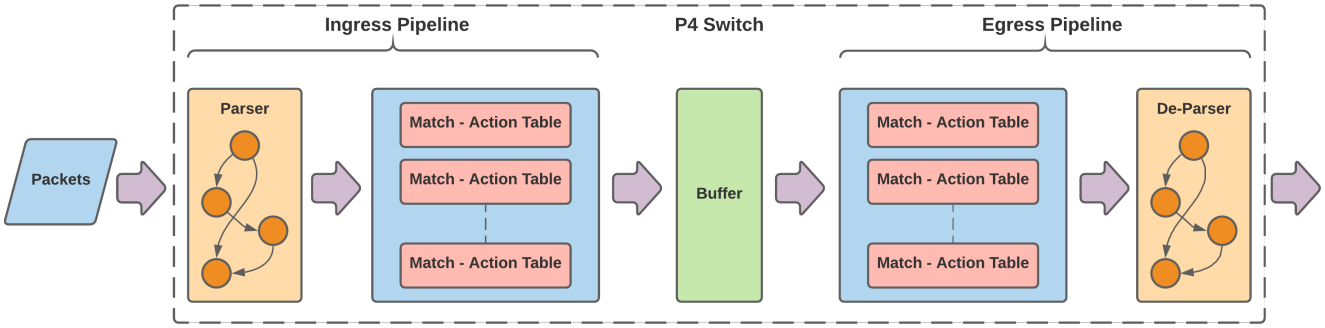


Figure 2.5: The basic P4 pipeline. Inspired by [p41c].

The P4 language [BDG+14, p4o] is an open source language that allows to program the logic of the packet processing pipeline in an actual switch or SmartNIC. More complete, it was designed with three main goals in mind: first goal is the reconfigurability, which describes the capability of changing the packet processing behavior of the switch at any time, as well as the ability to define custom actions that can be applied to a packet. The second goal is the protocol independence. In an empty P4 program there are no headers defined. The user then can define the desired headers. Afterwards, a parser needs to be defined that can parse the given headers. This design aspect allows the P4 language to be completely protocol-independent, which guarantees the implementation of any network protocol now and in future. The last goal is the platform independence. The official P4 language is designed in a logically higher level than the hardware so that any P4 capable product can be used. A compiler is used to translate the P4 language to a target platform. However, even if the P4 language is defined at a high level point, the design of the language is still close to hardware capabilities.

A P4 program consists mainly out of four main components which can be seen in Figure 2.6. The most important ones are the parser and de-parser, as well as the ingress and egress pipelines. In the ingress parser the packet header will be parsed. To simplify the parsing process P4 defines the parser as a graph. When a packet arrives, the first header gets parsed which is defined in the start state. After the first header is parsed, it is possible to make a transition to the next state, based on the previous parsed header values. In all states a packet header gets parsed and afterwards a transition to another state is possible. After parsing the last desired header, a transition to the end state can be performed to signal the parser that the parsing process is complete. Within the parsing process it is also possible to fill custom meta-data. Meta data is defined like a header and will be present on a per packet base. The meta data will be carried with the packet through the whole pipeline. An example for meta-data would be to store the physical ingress port number from which a packet originated. At the end of the pipeline there is the de-parser. The de-parser works like the ingress parser in reverse: it will take the headers and serializes the packet back to a bit stream which can then be transmitted towards another device.

Between the two parses, the ingress and the egress pipelines are located. Both are fully programmable and allow to implement the pipeline logic. As presented in Figure 2.6, a pipeline consists of a stage of tables. With these tables it is possible to match certain values against a parsed header. To describe the desired matching behavior, the header fields to match must be defined as key in the table. When matching against a value from the header, multiple options for the matching behavior are available. An

exact match, a longest prefix match and a ternary match can be selected. However, the matching tables alone are not sufficient. In order to make use of defined matchers, one has to install rules that apply them. When a rule is inserted, an action is also provided to the rule. An action then allows to perform certain modifications on the packet or to define the further processing within the pipeline. The actions must be predefined in the P4 program, but allow to pass variables when installing a rule into a table. An example for an action would be to define the egress port over which the packet gets transmitted when leaving the program. It is possible to define multiples tables and control the order of execution. Also, it's possible to influence the execution dynamically by using specific tables only on a dynamic condition.

Both the ingress and egress pipelines consist out of multiple user definable match-action tables and a control logic to define the execution of the table against the packet. Finally, there are two versions of the P4 language available. The older and simpler one is P4_14 [p41a] and the newer more advanced version P4_16 [p41b]. The differences will be explained in the next section.

2.4.1 P4_14

P4_14 is a more simpler pipeline version than P4_16. It has a fixed pipeline, like presented previously in Figure 2.6. The pipeline consists of a parser, an ingress pipeline, a packet buffer, an egress pipeline and a de-parser. The ingress and egress pipelines are programmable in the match-action style with the tables-action concept described above. The fixed layout of the P4_14 pipeline allows an easy entrance into the P4 world.

2.4.2 P4_16

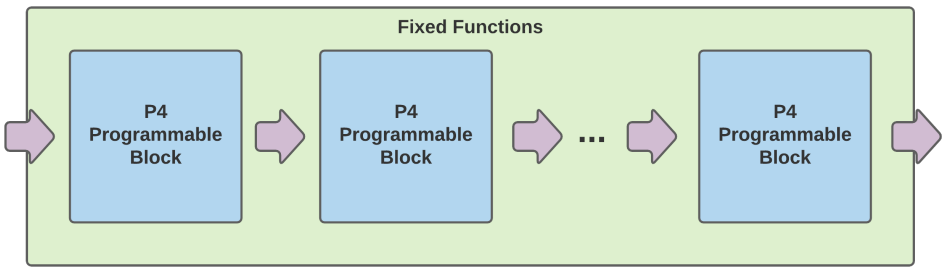


Figure 2.6: The P4_16 pipeline. Inspired by [BD17].

The P4_16 pipeline is the a incremental update to the P4_14 pipeline. The important change is that the main pipeline layout is not fixed anymore, like before. As in picture 2.6 is shown, the pipeline now consists of multiple P4 blocks within the data plane. Each block can be configured by P4 and then chained to create a pipeline. This allows to describe the pipeline in a more flexible way, while maintaining backwards compatibility with the P4_14 standard. The ingress and egress pipelines can be represented as two programmable P4 blocks. Finally, the redesign allows to open the P4 language to many new platforms.

2.4.3 Compiler

As mentioned before, P4 code is complied to a target platform. To support multiple different platforms the compiler is built with modular backends. The compiler first translates the P4 code into an intermediate representation. After the intermediate representation is generated, the selected backend compiler then generates the code for the specific target platform. Based on this modular approach P4 could be

compiled to any platform, as long as a backend compiler is available. This also allows P4 to be truly platform independent. For example, there are compilers for platforms like SmartNICs, switches and software switches available.

2.4.4 Barefoot Tofino

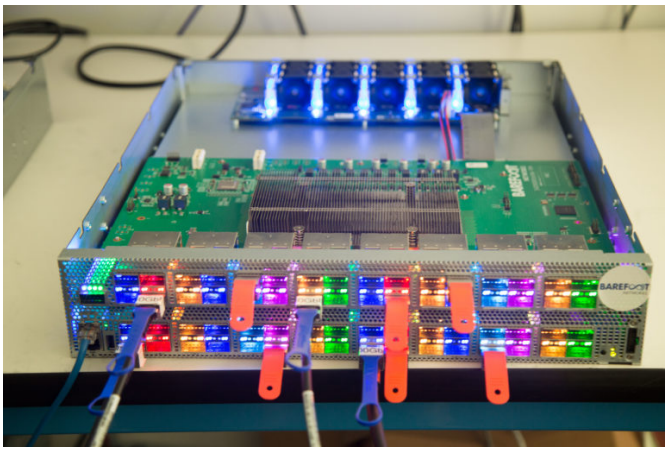


Figure 2.7: Image of a Tofino based switch. Source: [nex].

As mentioned before, the P4 language supports multiple targets. However, the first and most popular hardware switch that was available is the Tofino switching chip from Barefoot Networks [bfnb, bfna] which can be seen in Figure 2.7. The Tofino chip is built based on their proprietary Protocol Independent Switch Architecture (PISA). To support their switching chip, Barefoot provides a custom compiler. Their chips are available with up to 65 x 100 Gbit/s ports. The Tofino platform is up to date still the best way to go for a P4 compatible switch. As of today Barefoot network is still one of the major drivers for the continuous development of the P4 language. Currently most research activities in the area of programmable data planes build upon this switching ASIC [KUAh+20, JLZ+17, JLZ+18].

2.5 Summary

This section has introduced the most important technologies that are required to understand this work. The Internet was presented as a concatenation of multiple networks. The new emerging trends like SDN and NFV were introduced. Finally, the P4 language was presented. In the next chapter the related work to hybrid switches is presented.

3 Related Work

After some basics were covered in the last chapter, the focus will now be set on related work that was previously been performed on the topic about hybrid switches. Firstly, it will be examined whether the hybrid approach is feasible. Secondly, a first approach for a hybrid switch will presented. Afterwards the issues with rule dependency in hybrid switches will be discussed. Also, two approaches are examined and presented that try to solve those dependency issues. Finally, an analysis and summary will be provided.

3.1 Offloading Feasibility

In the introduction, the general idea of a hybrid switch was introduced. To summarize it up, the idea was to combine a hardware switch with a software switch. Then the software switch is used to store all rules and the hardware switch is utilized to carry the heavily utilized rules in hardware. The general idea relies on the assumption that it is possible to forward the most traffic with a small set of rules. This assumption must be proven to show the feasibility of hybrid switches. Recent work [SUF+12, WDF+05] has shown that the traffic of the Internet correlates to Zipf's law. This discovery proves the general idea that only a small set of rules is required to forward a major part of the Internet's traffic. Sarrar et. al. [SUF+12] have also confirmed this theory. They have investigated three traffic traces, one from a residential ISP, one from a transcontinental link (from the MAWI working group) and finally one from an European IXP. To map the traces against a common rule-set, they used a publicly available BGP full table to investigate the use case of routing. For the analysis they have calculated the traffic for each route of the full table per traffic trace. The results can be seen in Figure 3.1.

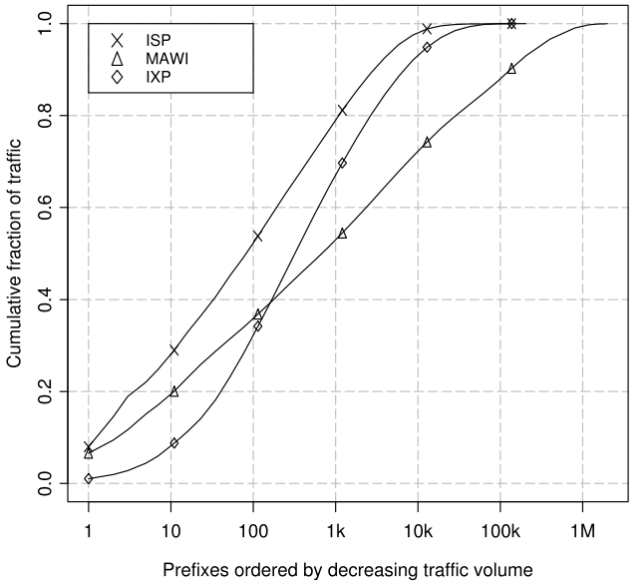


Figure 3.1: The most traffic can be forwarded by a small ammount of heavy hitter prefixes. Source: [SUF+12].

In the graph they have accumulated the traffic per prefix, sorted by traffic volume per prefix. This means that for example that in the case of the MAWI trace, the ten most active prefixes are responsible for 20 percent of the traffic volume of the overall trace. In all three traces, it can be observed that only a small number of prefixes are responsible for the most amount of the traffic. Those prefixes which carry the most traffic are called heavy hitters. The name comes from the idea that they are matched often due to the high volume of traffic. In case of the ISP and MAWI trace, only 10 heavy hitters carry at minimum

20 percent of the overall traffic. At 10k heavy hitters can carry over 90 percent of the traffic. The MAWI trace, in comparison, seems to be more distributed in terms of used prefixes. The authors assume that this behavior comes from the anonymization technique that was used on this trace, which has removed some granularity of the data.

Their research confirmed that it is possible to carry most of the traffic with a limited number of rules. Their result also agrees with additional works [SUF+12, LAW15, KARW16] that the general idea of offloading traffic is possible. This shows that the principle idea of a hybrid switch is feasible, at least for the use case of Internet routing.

During their research they have also detected that those heavy hitters are changing over time, which leads to new challenges, if a hybrid approach is used. This means a hybrid switch must be able to adapt to new situations, in case the traffic patterns change. To generally show and prove that hybrid switches can perform and adapt, Sarrar et. al [SUF+12] developed a strategy called TFO (Traffic-aware Flow Offloading) that enables the offloading function for a hybrid switch approach, based on the current traffic patterns. TFO will per presented in the next section.

3.1.1 Traffic-aware Flow Offloading

Traffic-aware Flow Offloading (TFO) [SUF+12] is one of the first attempts to implement a hybrid switch. TFO provides a strategy for offloading heavy hitters from the controller to a forwarder. Figure 3.2 provides the overall setup of TFO. The forwarder is referred as a hardware switch and the software switch is realized by the controller. The setup that is used to evaluate TFO can be examined in [SFU+10]. The design goal of TFO is to forward the most possible amount of traffic with the forwarder while reducing the update rate of the forwarder.

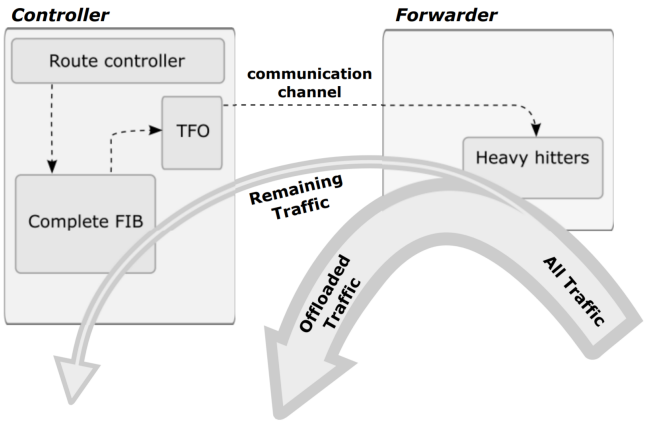


Figure 3.2: Setup of TFO. Source: [SUF+12].

To select which rules need to be installed into the forwarder, TFO measures the amount of traffic per prefix. This allows to detect the heavy hitter routes. To reduce the amount of changes to the offloading prefixes, they measure in different time bins. This allows them to reduce the change to heavy hitters that are fluctuating while allowing to detect flows that are trending to become a heavy hitter. In Figure 3.3 the four important steps for the offloading decision are provided.

Step 1 is to monitor the flows over different time bins - in the case of TFO ten seconds, one minute and ten minutes are chosen - to identify the top hitters that are offloading candidates. TFO sorts the flows by the traffic volume, now the *n* top flows are selected in a round robin style from the tree list. This selection procedure ensures that every top hitter from every time range has a fair chance to get offloaded. In step two the new selected offload candidates are compared to the currently offloaded ones. The output of the comparison are two sets of heavy hitters, one set with the ones to be installed and one set with the once to be removed. In step 3 the two sets are sorted. The sets are sorted, so that the highest valuable offloads

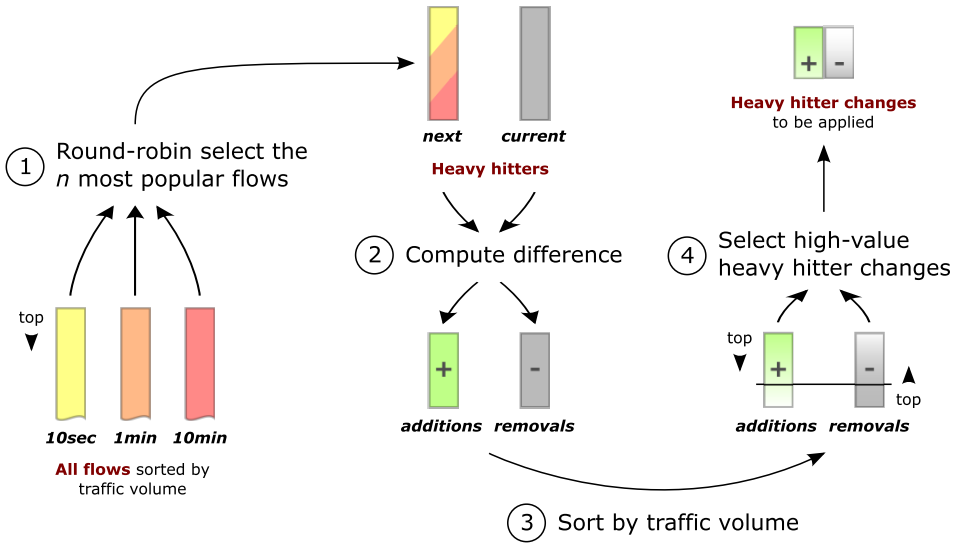


Figure 3.3: The four steps of the Traffic-aware Flow Offloading principel. Source: [SUF+12].

are preferred to be either offloaded or removed from offloading. Also, the sorting algorithm respects the timescale. In step 4 the selection of the actual change is made. The selection process decides if a new addition of a rule adds more value than the removed one. As threshold TFO requires that the new hitter must at least offload two times the traffic of the old one to be worth the effort to update the forwarder. The last two steps are the most important ones, because they allow TFO to weigh an update of a heavy hitter. This reduces the chance of install-and-remove cycles for heavy hitters.

According to their evaluation the TFO strategy works fine and achieves an offloading rate from 86 percent up to more than 99 percent, while also achieving the goal to keep the heavy hitters update rate low. The evaluation was performed against the three traffic traces that were presented in the section above.

However, from our perspective this approach has one major problem: it does not take care of any dependencies between two or more flows. This issue can lead to cases where TFO will cause invalid forwarding behavior within a network. This dependency problem is a general issue when dealing with hybrid switches. In the next section this dependency issue will be explained and multiple solutions are shown which enable the mitigation of the problem.

3.2 Inter-Flow-Rule Dependencies

In the last section it was concluded, that in flow offloading scenarios it is required to be aware of dependencies between flows. If conflicts between offloaded and a non-offloaded rules are not handled properly, situations can arise, where an incorrect forwarding behavior would be the consequence. In this section the problem will be explained based on an example. Finally, two approaches are presented which can deal with the inter-rule dependency on flow offloading.

3.2.1 The Problem

As mentioned before when using a hybrid switch approach, there is an issue with inter-flow-rule dependencies. To explain the problem, layer 3 routing is used as an example show case. In Figure 3.4 an example is presented. On the left side is a software switch located and on the right side a hardware

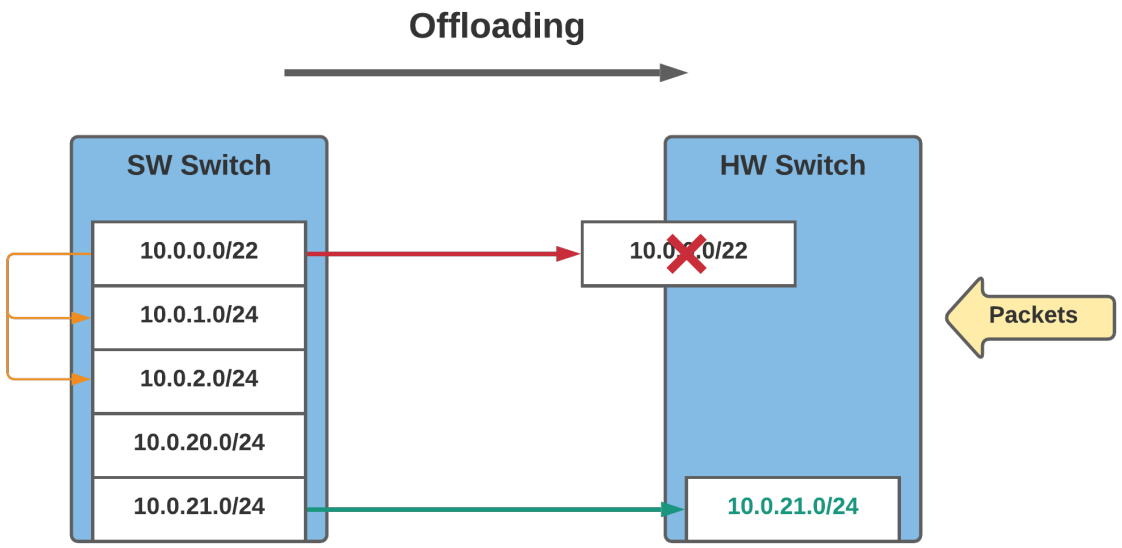


Figure 3.4: The dependency problem in hybrid switches. Prefix 10.0.0.0/22 can't be installed into the hardware switch due to its dependencies.

switch. The software switch will carry all available routing information while the hardware switch will only contain specifically selected prefixes. Traffic will also arrive on the hardware switch. First an example is provided which will refer to the default non-offloaded case. A packet will arrive at the hardware switch with the destination address of 10.0.20.1. On the hardware switch there is no entry that will match this packet, so the packet is forwarded to the software switch. Here a matching prefix is available: 10.0.20.0/24. In this case the correct prefix has matched, and everything is working as expected. Now a larger flow with destination 10.0.21.1 will pass through our hybrid setup. To reduce the load on the software switch, it was decided to offload the corresponding prefix 10.0.20.0/24 to the hardware switch. The traffic can now be matched and forwarded at the hardware switch. The forwarding behavior is still as expected.

Now to the problematic case: a larger flow with the destination 10.0.0.1 has arrived at the hybrid switch. Again, to reduce the load on the software switch it was decided to offload the corresponding prefix 10.0.0.0/22 to the hardware switch. For the current flow to 10.0.0.1 the forwarding behavior is still intact. However, as the prefix 10.0.0.0/22 was offloaded to the hardware switch, the forwarding behavior of two prefixes in the software switch is impacted. The prefixes 10.0.1.0/24 and 10.0.2.0/24 are contained within the prefix 10.0.0.0/22, as they are more specific. This leads to the problem that traffic for those two prefixes will now also be forwarded by the prefix 10.0.0.0/22 at the hardware switch, which is unwanted behavior. So, the prefix 10.0.0.0/22 can't be installed into the hardware switch. The prefix 10.0.0.0/22 has a dependency to the two more specific prefixes 10.0.1.0/24 and 10.0.2.0/24.

To successfully build a hybrid switch this problem must be solved to guarantee a correct forwarding behavior. To solve this dependency problem there are many approaches from which two are presented in the next sections.

3.2.2 Solution I: FIB Caching

Lui et. al. [LAW15, LAW13] introduced a hybrid approach for FIB (Forwarding Information Base) caching by using non-overlapping prefixes. This approach promises to solve the issue of inter-flow-rule dependency in the scope of layer 3 routing. They introduced the inter-flow-rule dependency problem as a cache hidden problem. The cache in this term is the device which is used for the offloading. In our

terms the hardware switch. The FIB in this case is the SDN controller. They clearly pointed out that FIB caching differs from normal caching mechanisms, because of the use of a longest prefix match for forwarding. This can lead to the case where a cached FIB entry can collide with a more specific entry that is not in the cache, which ultimately results in a wrong forwarding behavior. They provide an easy example which can be seen in Figure 3.5.

FIB entries and cache entries.

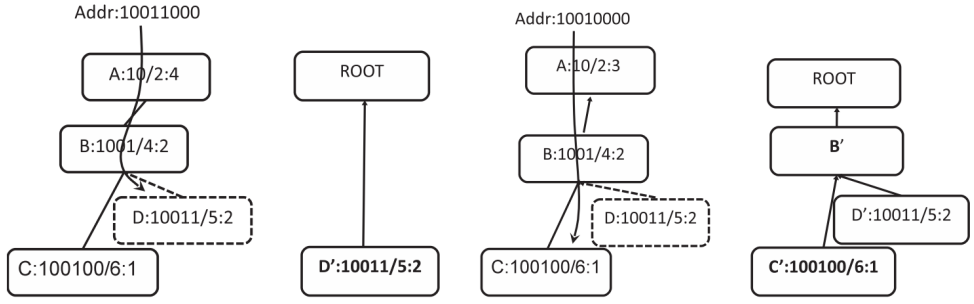
Label	Prefix	Next hop
(a) FIB entries		
A	10/2	4
B	1001/4	2
C	100100/6	1
(b) Cache entries		
B	1001/4	2

Note: The cache is initially empty and receives one entry upon the first cache miss.

Figure 3.5: Example FIB and cache. Source: [LAW15].

At first there are three Prefixes (A,B,C) in the FIB and it's assumed that in the first round the cache is empty. If a packet with destination 10011000 reaches the router, the router would look up its cache which is empty and make a lookup in the slow memory. Here a longest prefix match is performed which delivers rule B as result. The router installs rule B to the cache and forwards the packet to next hop 2. This leads towards the problem: the cache now hides entry C, because C is already contained in B. If a packet with destination 10010000 arrives at the router, the router would first perform a lookup in its cache which results in a match with entry B. So the router forwards the packet to next hop 2 which is incorrect at this point. In the FIB is a more specific prefix for this destination, which would forward the packet to next hop 1. This example shows that FIB caching is not trivial to solve, because it's not possible to move any entry into the cache without checking the indirect consequences.

To solve the cache hiding problem, they provide a solution called "FIB caching using minimal non-overlapping prefixes". The main idea is to only install prefixes that are not overlapping with any other more specific prefix in the FIB. If a prefix like B should be cached, it is only possible to install a part of B which is not colliding with C. In the example this would be 10011/5 which contains the largest part of B without overlapping with C. To illustrate and implement this, they use a Patricia Tries (which basically is a space optimized prefix tree i.e. Radix Tree). In Figure 3.6 this principle is provided.



(a) Generate prefix D in Full FIB

(b) Insert D' in FIB Cache

(c) Select prefix C in Full FIB

(d) Insert C' in FIB Cache

Figure 3.6: Example of generating non-overlaping prefixes in a patricia tries. Source: [LAW15].

In (a) it can be observed that prefix C is contained in prefix B because C is a child of B in the Patricia tries. The implications are that only leaf nodes in the tree are non-overlapping prefixes. In the situation when a packet arrives which would resolve to cache entry B, there would be a cache-miss, as B is not

directly cacheable. To cache the possible part of B, a new node D is introduced which is non-overlapping with B. This node can now be installed into the cache (b) without any conflict. If now a packet arrives (c) which would match prefix C, there would be a cache miss with the consequence that the router needs to lookup the prefix in the FIB which now leads to correct forwarding behavior. The authors also mentioned an interesting side effect of this approach. Since all prefixes in the cache are now non-overlapping, no priority between them is required when installing those into the TCAM of a switch. The lack of a priority enables faster update rates of the TCAM because no reordering must be performed.

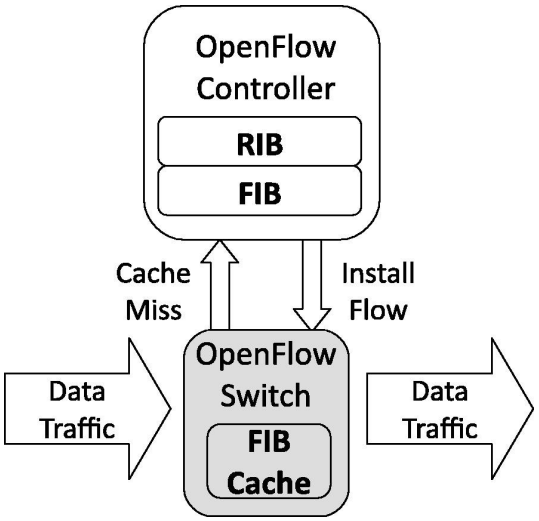


Figure 3.7: Setup for the Mininet simulation. Source: [LAW15].

To prove their approach, they implemented a prototype in a Mininet [min] simulation (see Figure 3.7). The cache device was an Open vSwitch [PPK+15], an OpenFlow controller was handling the cache misses, the full FIB, and the control logic. When the controller starts it calculates an initial cache set to be installed into the cache. After the initial setup, a cache replacement strategy is needed to handle dynamic changes to the cache. Most important is the case when the cache is full and one entry needs to be removed, in order to cache a new one. Classical cache replacement strategies were not adequate for this use case due to the limitations of the OpenFlow protocol. An LRU (least recently used) strategy would require the controller to monitor the idle timeout of a cache entry, to see which flow entry times out first. The issue is that OpenFlow does not support to readout the current value of the idle timeout. It only notifies the controller if a timeout occurs and the entry gets removed. This behavior does not allow to identify a cache entry which can be replaced, if the cache is full. Another popular strategy is the LFU (Least-frequently used). The idea is to replace the one entry in the cache which is used at least. To implement this, a counter is required to count the packets for each flow entry, which OpenFlow can provide. The issue is, to select the least used flow entry, all counters of all cache entries need to be queried. The problem on this approach is that most OpenFlow switches are not capable to deliver the required counters in an adequate timeframe. Another drawback is that the counter for every flow entry needs to be queried individually, so the required time to query is linear to the cache size. Since none of the standard cache replacement strategies work in the OpenFlow context, they provided their own cache replacement strategy.

Their provided approach utilizes the idea of increasing the timeout for cached entries that are more frequently used/hit. To determine an optimal default timeout, they investigated a 24-hour traffic trace and came up with 43 seconds, which is used as initial timeout for a new entry. When an entry is now installed into the cache, a second timer is started which counts down from 60 seconds. If now an entry gets removed after 43 seconds and the entry needs to be installed into the cache again before the 60 second timer reaches 0, the timeout of the rule can be increased. The idea behind the concept is, if an entry is reinstalled more often it is also carrying more traffic, so it would be beneficial to keep it longer

in the cache to reduce the overhead of re-installing that rule. Of course the second timer also needs to be increased, because if the cache timeout is greater than the second countdown, the counter is at zero before the rule times out which then leads to installing the important rule again with only the default timeout, which would be inadequate for a high hitter entry.

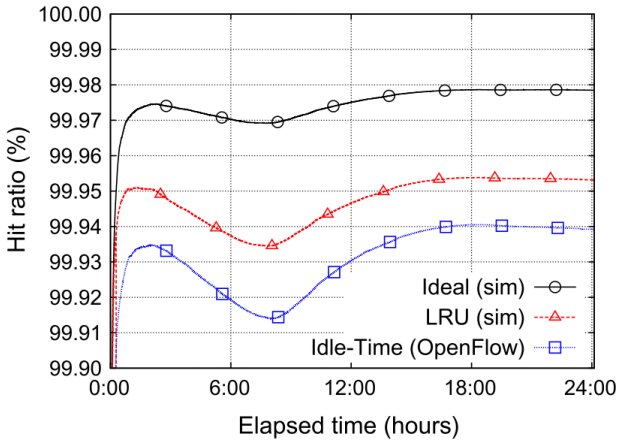


Figure 3.8: Result of their timeout approach with OpenFlow Source: [LAW15].

The evaluation in a simulator can be found in Figure 3.8. Their cache replacement strategy performs almost as good as a LRU strategy, which they've simulated to compare their result. Also, they simulated an optimal cache entry setup to observe what could be possible in a best-case scenario. The result proves that their approach enables a high hit ratio in the cache over an 24 hour trace.

Their results seem to be promising and show that it is possible to maintain a high hit rate in the cache. However, they did not evaluate their approach with real hardware and massive traffic rates. Also, they did not perform any research to the relation between a high cache hit rate and the bandwidth of the flows. So, a simple yet interesting example is how much bandwidth the cache could actually serve, if there are lots of low bandwidth, but high pps flows and some relatively low pps flows with high bandwidth due to larger packet sizes. Is the metric hit-rate even efficient for the use-case? Yes, a high hit rate is a good indicator for using a resource in an efficient way, but there are other important aspects like the cached bandwidth.

3.2.3 Solution II: CacheFlow

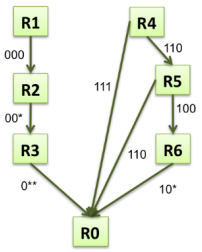
In previous sections the offloading/cache mechanisms were discussed with focus on offloading on OSI layer 3 for routing. However, in modern SDN networks the forwarding is mostly built upon on a per flow based model including attributes from layer 1 up to layer 4 of the OSI model. These additional layers add more potential for inter-flow-rule dependencies which become even more important when flow offloading is considered. To deal with those more complex and longer dependency-chains a more detailed approach is required.

Katta et. al present CacheFlow [KARW16] which investigates the dependency between flow rules. They present an approach to deal with these complex dependencies to enable flow-rule offloading. They've pointed out that there are two main reasons why these complex dependencies could arise. The first one are Partial overlaps. As with layer 3 destination-based forwarding, it's possible that one rule contains another. In layer 3 this is straight forward, because there is only one header which could overlap. When considering multiple headers, more complex dependencies can arise over multiple fields. For example, one rule matches the destination of a host and another rule matches the same destination with a specific protocol and a higher priority. Even the priority between rules can lead to more complex dependencies, because lower priority rules cannot be offloaded without considering the higher priority

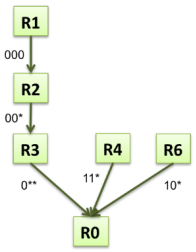
rules, which then could be hidden by the cache. The second source of complex dependencies can arise from policy composition. SDN hyper-visors or platforms like Frenetic [FHF+11], Pyretic [RMF+13], Co-visor [JGRW15] enable multiple applications to define policies for a given network topology. The rules from each application can be dependency-free, but if the rules are combined into the same forwarding hardware, complex dependencies could arise.

Rule	(dst_ip, dst_port)	Ternary Match	Action	Priority	Weight
R1	(10.10.10.10/32, 10)	000	Fwd 1	6	10
R2	(10.10.10.10/32, *)	00*	Fwd 2	5	60
R3	(10.10.0.0/16, *)	0**	Fwd 3	4	30
R4	(11.11.11.11/32, *)	11*	Fwd 4	3	5
R5	(11.11.0.0/16, 10)	1*0	Fwd 5	2	10
R6	(11.11.10.10/32, *)	10*	Fwd 6	1	120

(a) Example rule table



(b) Incremental DAG insert



(c) Incremental DAG delete

Figure 3.9: Example construction of an incremental dependency DAG. Source [KARW16].

To understand and represent the dependencies between those rules, they proposed to build a Directed Acyclic Graph (DAG). In the DAG each vertex represents a rule and the directed edge between two vertices presents the dependencies. An example can be seen in Figure 3.9. Here in (a) an example rule set is provided and the corresponding dependency DAG. In the DAG it's now possible to determine which rule has which dependency. Important is that R0 is the root of the graph which connects all independent sub-graphs into one graph. For example, rule R3 in the graph has some dependencies to rule R2 and R1. If R3 would be selected to be offloaded or cached, rule R1 and R2 would be masked by the cache. At this point some techniques are required to prevent the masking. However, there is one more requirement to the graph.

Flow-table sizes are growing and changes to them become more frequent. In consequence it's infeasible to generate a new DAG after each update of the flow-table. To reduce the effort, an incremental DAG is required which only performs the partial changes that are required for the transition between the new and old graph. The transition from (b) to (c) visualizes the changes made due to the deletion of R5. The algorithm that calculates the incremental change only updates the right side of the graph and does not modify anything on the left side. This allows to perform quick on-the-fly changes to the graph on flow rule updates. More about the incremental update of the graph can be read in the paper [KARW16].

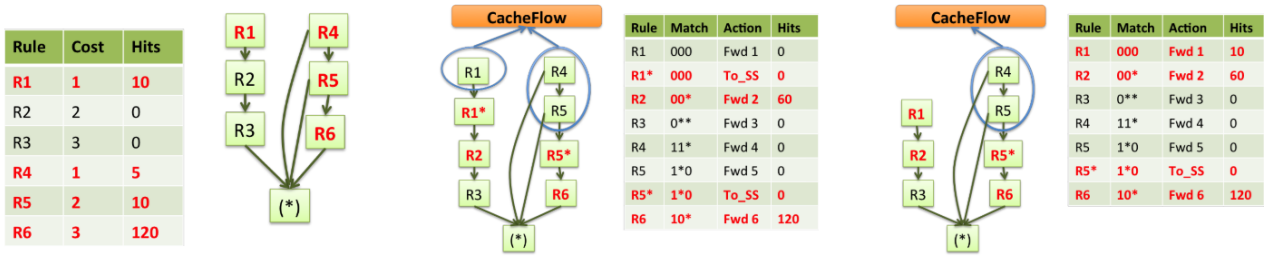
Now the dependency graph can be built and updated at any time. To enable caching, algorithms are required to install rules into the cache without violating any dependencies. In the next subsection two different approaches are shown and finally one that combines the best of both worlds.

Dependent-Set

The first approach to solve the dependency dilemma is to install all dependent rules into the cache. However, since the cache size is limited the installation comes with a cost in form of required size in cache. The first approach was to build a strawman algorithm to prevent low weight rule caching. For this every rule received a cost metric, which represents the number of rules that need to be installed with them. In Figure 3.10 (a) the cost behavior can be observed: rule R6 has a cost of 3, because in order to function correctly, rule R4 and R5 also need to be installed, which then results in the installation of three rules into the cache. To also weigh the rules against each other to determine the importance of them, a hit metric is added to show how much traffic will be forwarded by the given rule.

Finally, now a decision needs to be made which rules can be installed into a cache with limited space. This optimization problem is NP-hard and can be derived from the densest k-subgraph problem. Since no approximations are known that run in polynomial time, the authors decided to use a greedy heuristic.

In each greedy iteration the algorithm tries to find a set of rules that maximize the ratio between weight and cost. The weight would be the hit rate and the cost the number of rules that need to be installed. The algorithm then runs n rounds until the sum of the selected rules reaches the cache size target. In Figure 3.10(a) the results of the algorithm can be seen for a cache size of 4.



(a) Dependent Set Algo. (b) Cover Set Algo. (c) Mixed Set Algo

Figure 3.10: The three dependency resolve strategies of CacheFlow. Source [KARW16].

Cover-Set

The approach of installing all dependent rules into the cache can quickly become expensive, since the cache has limited space and the installed dependency rules might not have a high hit rate. This leads to inefficient cache utilization, because it is not guaranteed that the dependency rules have a high hit rate. To solve this problem an approach to install the important rules into the cache with the minimum cost to respect their corresponding dependencies is required. The idea is to compute a new rule out of the dependencies, which then leads to send the packets for the dependencies up to the slow device / switch. In Figure 3.10 (b) the idea can be observed. Rule R6 has two dependent rules, so the installation of rule R6 would require the installation of three rules with the old approach. To reduce the number of installed rules into the cache, a new rule R5* is calculated which matches all traffic that matches the dependent rules (R5, R4). The action of R5* is then to send the packet to the slow path, which packets normally take when a cache miss occurs. By calculating this new rule R5* it is now possible to install R6 with a cost of 2 instead of 3, because only two rules need to be installed into the cache. This saving effect even becomes handier if the dependency chains are getting longer.

Mixed-Set

As presented before, Cover-Set enables to minimize the amount entries being installed into the cache. Unfortunately, Cover-Set is not always the best solution. The reduction of the cache size comes with the cost of using more slow path bandwidth. In Figure 3.10 (c) the offloading of R2 would be more efficient when using Dependent-Set, since it would serve more traffic directly with the cache compared to Cover-Set. Also, it reduces the load on the slow path. Both solutions have an equal cost in terms of cache size. So Dependent Set would be the better choice in this case. From this observation it can be learned that the caching strategies also require to be aware of different available techniques to find an optimal solution. In order to make use of this observation, CacheFlow weighs Dependent-Set and Cover-Set against each other on every offload of a rule, in order to make use of the optimal caching strategy for each situation.

Prototype

To test their approach, they have implemented CacheFlow in a prototype by using Ryu [ryu] which enables an OpenFlow interface in transparent mode. This allows the CacheFlow Controller (Cache Master)

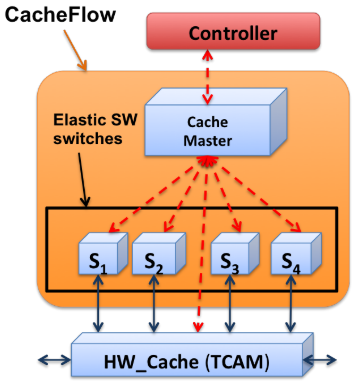
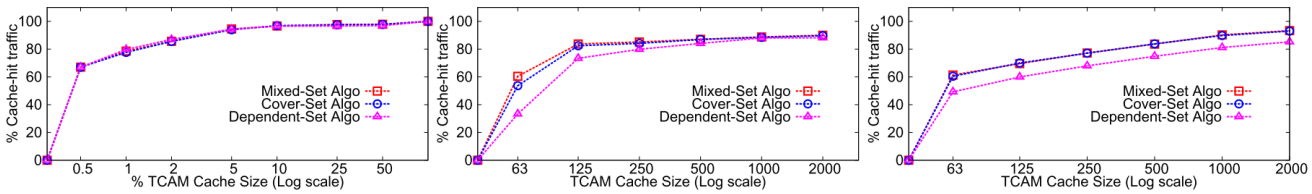


Figure 3.11: Prototype design of CacheFlow. Source [KARW16].

to appear like a single device towards the SDN controller. The basic overview of the implemented architecture is depicted in Figure 3.11. The hardware switch is used as cache and for the cache misses a Open vSwitch software switch is used. To test their approach, they have used three traffic traces, one from an IXP, from Stanford Backbone and the CAIDA trace from Equinix. They measured all three cache algorithms independently for each trace. The results can be examined is Figure 3.12. It can be observed that only in one trace all three algorithms perform equally. In the other two traces, it can be observed that Dependent-Set performs worse that Mix- and Cover-Set. However, in each trace with an appropriate cache size at least one algorithm has achieved over 90 percent cache hit rate. Finally, they have measured the latency of the cache. On a cache hit they achieved 0.71 ms and on a miss 0.81 ms, which means that a miss costs around 0.1 ms. However, in their paper they have described their measurement-method as the average round trip time of a ping packet. This seems like a quite unprecise measurement, if the implication is right that a simple ping tool was used. As the ICMP request and response mostly do not have the highest priority on an operating system, which could add latency to the measurement packet which is not introduced by the data plane.



(a) REANNZ IXP switch (b) Stanford backbone router (c) CAIDA packet trace

Figure 3.12: Evaluation of Cacheflow. Source [KARW16].

3.3 Other Use cases for Hybrid Switches

In the last sections mainly the uses cases of general flow offloading were discussed. However, there are at least two other use cases that are worth to be mentioned. In this section two new use cases are presented.

3.3.1 ShadowSwitch

In the previous section, it was mentioned that a common problem with SDN enabled switches is the lack of table space. However, there are also other shortcomings. Another issue is the flow installation

time. This describes how much time the switch requires to actually install a given rule into its hardware memory. The setup time of new flows is also limited, the setup time may not be linear due to internal optimization processes within the hardware. To address this issue, Bifulco et. al. [BM15] present a hybrid switch concept. They have observed that jitter in flow installation time is not present in software switches. The installation time within a software switch seems to be linear. Also, the software switch enables a higher update rate compared to the hardware one. To utilize the benefits of the software switch, they proposed a hybrid switch called ShadowSwitch. The design of ShadowSwitch is presented in Figure 3.13.

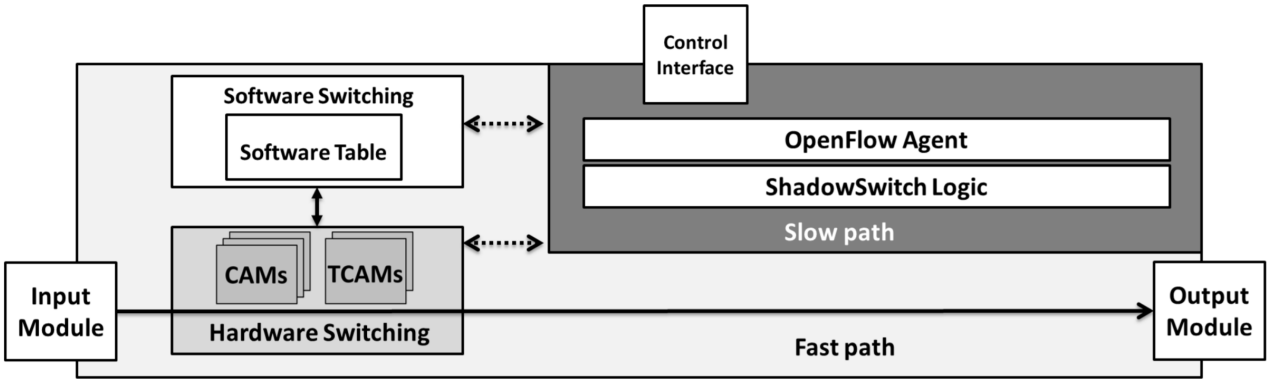


Figure 3.13: Architecture of ShadowSwitch. Source [BM15].

ShadowSwitch consist of a hardware switch and a software switch that are connected together. At first, in order to reduce the time a new flow installation takes, ShadowSwitch installs new flow rules only to the software switch. This allows ShadowSwitch to achieve a linear flow setup time. When the flow rule is installed into the software switch, ShadowSwitch will move the rule to the hardware switch in the background.

They also mentioned that ShadowSwitch is also facing the issue of incorrect forwarding behavior due to the higher priority of the hardware switch. It was mentioned that this was solved by using strict priority. A prototype based on a hardware OpenFlow switch and a software OpenvSwitch, which runs on a commodity x86 server, was implemented.

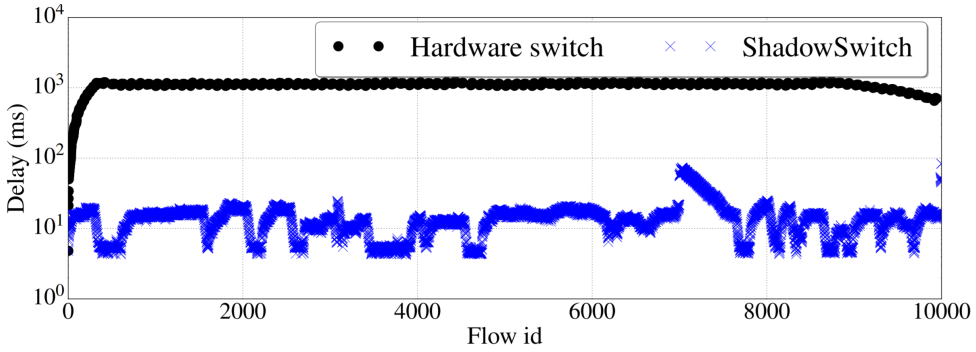


Figure 3.14: Evaluation of ShadowSwitch based on flow installation delay. Source: [BM15].

Their results can be seen in Figure 3.14. The provided graph shows the setup delay of ShadowSwitch, compared to the normal setup time of a single hardware device. The presented results prove that a hybrid switch can also be used as a kind of accelerator for hardware devices. Finally, it was pointed out that ShadowSwitch is ideal to be used for reactive flow installation scenarios, because the limiting factors like update rate and installation time are no constraints anymore.

3.3.2 OpenState

In software-defined-networking the main design goal is to decouple the control plane from the data plane. This allows to implement the logic on a logically centralized controller. However this approach can lead to increased overhead for simple network logic, since every function must be handled by the controller. There are many use cases where it would make more sense to handle the logic on the device itself. For example, MAC address learning or stateful firewalling. One approach to bring logic back to the data plane is OpenState. OpenState [BBCC14] is a minimal extension to OpenFlow which allows to implement Extended Finite State Machines (XFSM) in hardware. This allows to realize logic that can be handled in a state machine to be implemented within the hardware to reduce the controller overhead. An example would be port knocking in a firewall, shown in Figure 3.15.

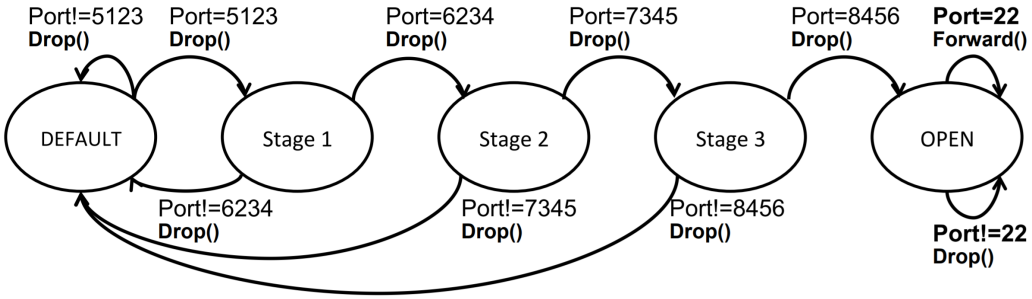


Figure 3.15: Example Finite State Machine for firewall port knocking. Source: [BBCC14].

The state machine in Figure 3.15 represents the following logic: to access port 22, four other ports must be knocked before to allow the communication. To implement this state machine in hardware, two tables within OpenFlow are required. The fist one is the state table which keeps track of the current state of a flow and second a XFSM table is required which describes the actions for each state. The XFSM table requires an extension to OpenFlow for the next state field. The new field allows to perform a transition within the state table towards a new state for a given flow. An example for a state and XFSM table can be seen in Figure 3.16.

The main concept seems to be promising, but again the issue is the lack of flow table space in hardware switches. Software switches again do not have this constraint of table space, but do instead suffer in terms of performance on high forwarding rates. To solve this issue, Roberto BIFULCO patented [BM19] the idea of using a hybrid switch design for the use case of Finite State Machines in the Data Path. They propose to use a hardware switch as an offloading device for a software switch. Their approach is to setup the state table completely in a software switch at first. Then they implemented an algorithm that transfers parts of a state machine into the hardware, if the load is starting to exhaust resources within the software switch. The design requires to have one state and one XFSM in hardware and software, which can be determined in Figure 3.17.

However, there is a patent on this technology and it is important to understand the scope of the patent: it only covers the use case of Extended Finite State Machines. So, the idea of a hybrid switch in general is not be covered by this patent.

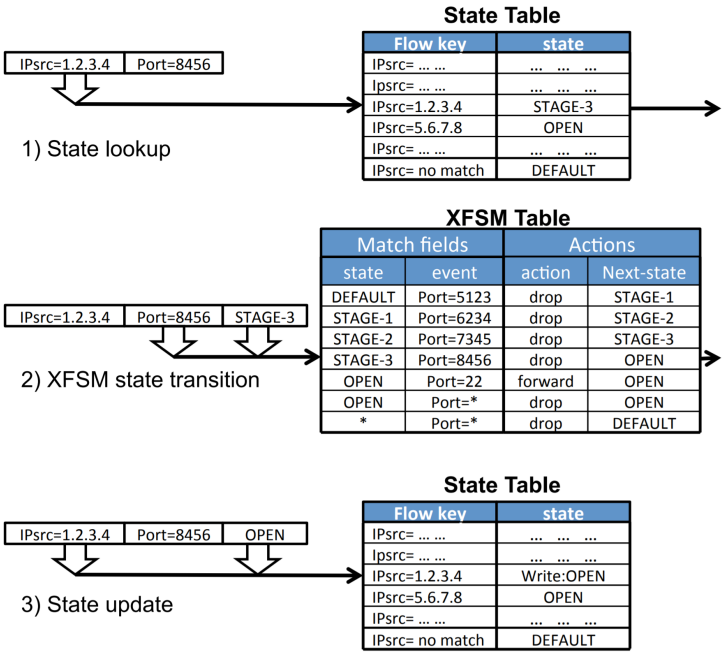


Figure 3.16: Visualization of the State and XFSM Table for the firewall example. Source: [BBCC14].

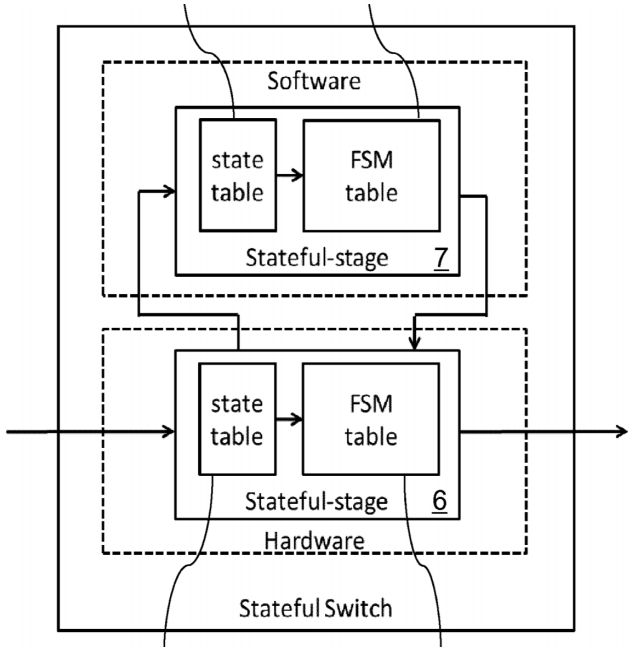


Figure 3.17: Hybrid Setup consist ouf of two tables in the software and hardware device. Source: [BM19].

3.4 Summary & Analysis

Until now this chapter has provided an overview over different approaches on hybrid switches and how they function. It was shown that all concepts were developed based on different needs, for example the lack of table space or missing performance in terms of flow updates within the hardware. Also, it was worked out that hybrid switches come with their very own challenges. The biggest challenge is to handle inter-rule-dependencies. The problem is one of the most complex ones for hybrid switches, since if this problem is not correctly handled, the forwarding behavior within a network will be disrupted. To solve these problems, multiple solutions have been provided, like Cover-Set or non-overlapping-prefixes. Those and other solutions allow to resolve these dependencies in a sufficient and efficient way.

However, on the other hand it can be seen that the focus of the previous work relies mostly on the control plane site, by solving the inter-rule-dependencies. To prove their concepts, the previously mentioned authors have each implemented some prototype of hybrid switches, but mostly in simulated environments. If a prototype of real switching hardware was developed, they mostly built up on OpenFlow enabled switches. Due to the constraint that comes with most of those devices - like missing flexibility, lack of programmability or performance issues while interacting with those OpenFlow switches - they were not able to develop and implement a state of the art data plane for hybrid switches yet. Astonishingly, there is almost no dedicated work performed on data plane performance on hybrid switches.

As the technology has evolved over the years and many new technologies have been developed, it should now be the right time to rethink the data plane of hybrid switches. One could even go that far and question the dependency to OpenFlow enabled devices at all. New P4-enabled switches leverage full programmability while still achieving very high performance.

This work will address the lack of research on data plane of hybrid switches. Based on the latest technology, a new data plane for hybrid switches will be developed. By utilizing modern hardware like P4-enabled switches and the DPDK, we should be able to reach forwarding performance up to 100 Gbit/s and more. Also, with a complete redesign it should be possible to introduce true programmability and to find new adequate solutions for problems of the past, like performance issues on querying hardware counters. Finally, the past has shown that involving the controller within the forwarding is not sufficient for reaching the maximum performance. The redesign allows us to create a complete dedicated data path, where no packet needs to be forwarded by the controller.

In the next chapters the new design will be developed, implemented, and finally evaluated.

4 Design

In this chapter the design for our hybrid switch is provided. Before the design is presented, a quick recap is given. Current switches can perform various tasks in networks like flow-based forwarding or routing. Those SDN-enabled switches are designed to perform their tasks at line rate to deliver high performance. By using commodity off-the-shelf switching ASICs, which are mass produced, they can deliver all those functionalities at a low price target. The issues here are that these switches have limited hardware table space. Since modern networks make use of many concurrent flow rules or routes, it's likely that a given rule set can exhaust the table space of a switch. Another problem with these ASICs is the fixed hardware after manufacturing, which reduces the ability to implement new features later on. Also, the hardware resource allocation is mostly fixed. So, it is not always possible to remap the table space of a unused feature to another feature that is heavily used. If a switching ASIC does not have enough table space or does not support a given feature, the only way to move forward is to replace the switch with a more capable one.

To tackle the issues with the table space, a caching architecture can be used. While the ASIC can deliver high forwarding performance at low cost, it cannot scale in terms of the table space, since it is a fixed-size resource. To utilize the limited resource efficiently, the switch is transformed into a high-performance cache. This cache should only be used for the most important rules in the network, which handles most of the workload. Most likely there are many rules left that may not carry as much traffic, but are also important in order to the forwarding to function properly. A naive approach would be to use a second hardware switch behind the first one. Afterwards all rules that couldn't be installed into the first switch can be installed into the second one. This approach would also be limited, since the table space is again fixed and then another switch must be added and so on.

A countermeasure to all those scaling problems is the software layer. In the past several years, standard server hardware with networking software has evolved to a point where servers are capable of line rate performance on packet forwarding. They cannot deliver the kind of forwarding performance like an ASIC, but they have a key advantage over the ASIC, which is lots of system memory that enables the ability to handle large tables to store forwarding rules. In the end both the server and ASIC have their disadvantages that the other one can overcome, so if both are combined, they should be able to deliver high performance with plenty of table space for rules. The switch ASICs have only to be able to handle the main rules that carry the most workload. All other functionality can be realized by the software switch on a standard server.

Another concern with current switch ASICs is that the resource allocation is fixed and the packet pipeline can't be adjusted after manufacturing. This situation limits the cache device to a fixed function set that cannot be adjusted to future needs or new protocols. To enable switches to be more flexible, a new language called P4 [BDG+14] was specified which enables to program the complete forwarding pipeline of a given switch. This now allows to allocate the resources of the switch accordingly to the use case and implement new functionality in the future, without the need of upgrading the switch itself. The combination of all the technologies mentioned above should enable us to create a new hybrid switch called P4HC - P4 Hybrid Cache. By combining the forwarding performance and flexibility of a fully programmable switch as a cache with a fully programmable software switch as a main forwarder, table size should not be a concern anymore and also future upgrades or changes can be easily performed due to the fact that the whole stack is fully programmable.

In this chapter the design of P4HC is introduced. First a high-level view on the architecture is presented. Afterwards the two main components are explained in detail.

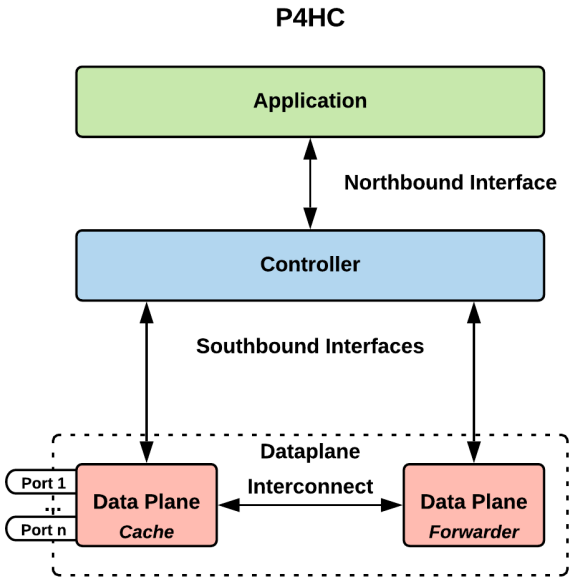


Figure 4.1: Design of P4CH: One controller and two data planes are spliced together to enable flow rule caching.

4.1 High Level Architecture

Figure 4.1 describes the main components of the P4HC design. P4HC consists of a single controller and two data planes. The architecture was designed with the classical SDN architecture in mind. The idea of the strict control and data plane separation - by utilizing well defined APIs - enables full modularity in P4HC, so that any component could theoretically be replaced. The P4HC controller provides a northbound API which enables the communication with a network application that sends flow rules to be installed into the data plane. The P4HC controller then implements the logic that is required to control two data planes in a way that one acts as a cache device for the other. Both data planes are controlled over a well-defined southbound API which enables P4HC to switch between different data planes. To prevent any confusion in the future, it's key to understand that the main goal for P4HC is to be transparent to any other device in the network. This means, a connected device or a network application will not be aware that there are two devices involved. The second design goal is to keep full data plane programmability to enable P4HC to be compatible with any protocol now and in the future.

4.1.1 Data Plane Splitting

As mentioned, P4HC consist of two data planes, where one data plane acts as a cache for the other. This idea can be reviewed in Figure 4.2. Here the forwarder and a cache data plane can be examined. Both data planes spliced together act like one, so that the network subscribers cannot notice the two-stage setup. The first data plane is the forwarder data plane which should hold all flow rules. To provide enough table space, the forwarder is fully implemented in software. While flexibility is thereby not an issue, the challenge is to maintain full line rate performance. The cache data plane is placed between the forwarder and the network subscribers. This cache is represented by a hardware switch which has plenty of physical ports to connect to the subscribers and allow full line rate performance for the installed flow rules. The main challenges here is that the table-space on the hardware device is exceedingly small in comparison to the software forwarder. Also, programmability of the hardware is limited, mostly.

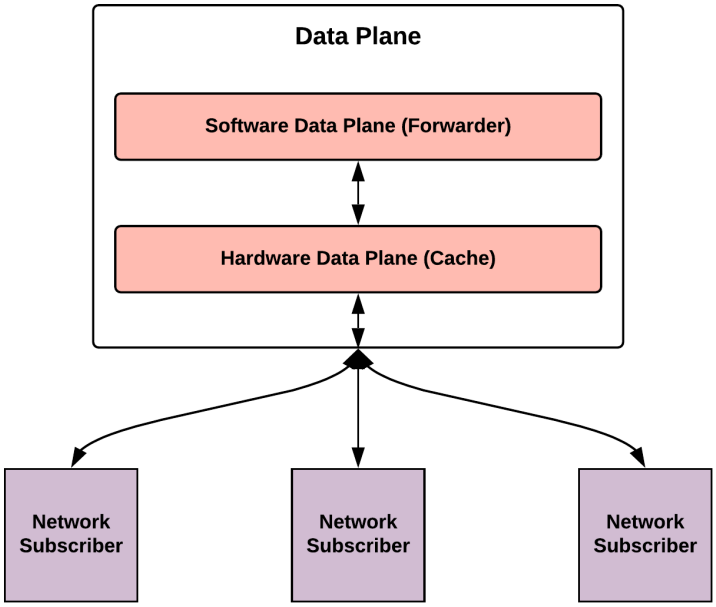


Figure 4.2: Shows how cache and forwarder are staged to enable data plane caching.

Since one of the main design goals of P4HC is to maintain full data plane programmability, a common language is required to describe both data planes in hardware and software. The selection for P4HC is the P4 language. P4 enables to fully define a forwarding pipeline, while being completely protocol-independent from the language point of view. P4 pipelines can be compiled to hardware targets or software targets according to its specification.

To further understand the interaction between the forwarder and cache, an example is provided. A new rule is only installed in the forwarder at first. Given a packet is then received at the hardware cache, the corresponding flow rule is not yet installed into this cache - a cache miss is the consequence. The packet is now sent from the cache to the software forwarder, where the packet has a corresponding flow rule to match against. The packet is matched and tagged with its destination port of the switch and forwarded back to the hardware cache, which will then forward the packet to the specified port as determined by the software forwarder.

4.1.2 Control Plane

In P4HC the controller brings all together, while coordinating both data planes it must also hide the separation of the two data planes to the network, so that they act transparently like a single switch. The classical controller has only one interface to one switch. In P4HC one switch consists out of two data planes, so the controller has to handle two interfaces. One connecting to the forwarder data plane and one towards the hardware cache. To reduce the complexity of those two interfaces, they should be implemented in the same interface language and have common functions like adding a flow or removing a flow, for example. This provides modularity to the controller, since the data planes can be theoretically exchanged with minimal effort.

To coordinate the forwarder and the cache the controller must implement the off-loader capabilities. All rules that the controller receives from the network application should be installed in the forwarder, since the software data plane has plenty of table space available. The forwarder has the issue that it cannot handle as much traffic as the cache. So, the controller must decide when to offload a flow rule from the forwarder to the cache. When the controller decides to offload a rule into the cache, it must

take in consideration that flow rules may have dependencies to other rules. In order to maintain the correct forwarding behavior, the controller implements an algorithm that can solve those dependencies and thus maintain the correct forwarding behavior when offloading a flow rule. Finally, the controller needs a way to determine when to offload a rule. So, the controller must have the capability to inspect the load on each single rule to decide when to offload a rule into the cache and when to remove a rule from the cache again.

4.1.3 Sum Up

In this section a short introduction to P4HC was provided. In the next sections the main components (data plane and controller) are drilled down to provide the exact details. However, the general principle of flow rule caching/offloading is valid and functional for any kind of flow rules. As flow rules become more complex, also their dependencies can occur between multiple rules which makes the caching even more difficult. Since the goal of P4HC is to show that flow rule offloading with high performance hardware while preserving full flexibility, is in fact possible, the complexity of flow rules must be limited. The decision was made to show the feasibility based on OSI layer 3 IPv4 routing. From our perspective routing is one of the most common use cases in large networks today and one of the costly ones. Capable routers from various network vendors are expensive and have low port density in comparison to switches. Today's switches can handle IPv4 routing, but have not enough space available in order to handle a full Internet routing table. So, this use case should be perfectly suitable to verify if a switch used as a cache combined with a software forwarder can compete with a large router and deliver the needed performance. However, since our design also lays focus on full programmability, it should be possible to adapt P4HC to any other use case or flow rule types. Since the data plane is fully programmable, it should be able to handle any kind of traffic.

4.2 Data Plane

In this section the data plane design of P4HC is presented. First a quick introduction into a layer 3 routing pipeline is provided. Afterwards it will be discussed how to slice the single pipeline into a dual device pipeline. To implement the layer 3 routing pipeline into a network, layer 2 forwarding must be implemented too, which requires additional modifications to the overall pipeline. After presenting the initial pipeline design, it will be worked out how to generate metrics of the data plane's traffic for the controller. Finally we introduce how the P4 language is utilized to describe our dual stage pipeline and what the current challenges and limitations are.

4.2.1 L3 Routing Pipeline

The goal of the pipeline design of P4HC is to implement layer 3 routing. Traditional routing takes place on a single device. Within this device there is a forwarding pipeline that contains a longest prefix match table to perform the actual routing decision. In Figure 4.3 this classical approach is depicted. Every other part of the pipeline has been cut off for simplicity.

The Figure describes an abstract view of a routing-enabled device with four ports and a single lookup table. In this case packets will arrive on port 0. When a packet is received at port 0, it will be matched against the longest prefix table. This table contains all available routes and the according next hop. The next hop is the identifier of the port through which the packet will be transmitted. For example, a packet with the destination address 10.10.0.2 is received at port 0. Then a LPM lookup will be performed with this destination address. The result would be "Prefix: 10.10.0.0/24 Next: 2" which then would lead to sending the packet out through port 2.

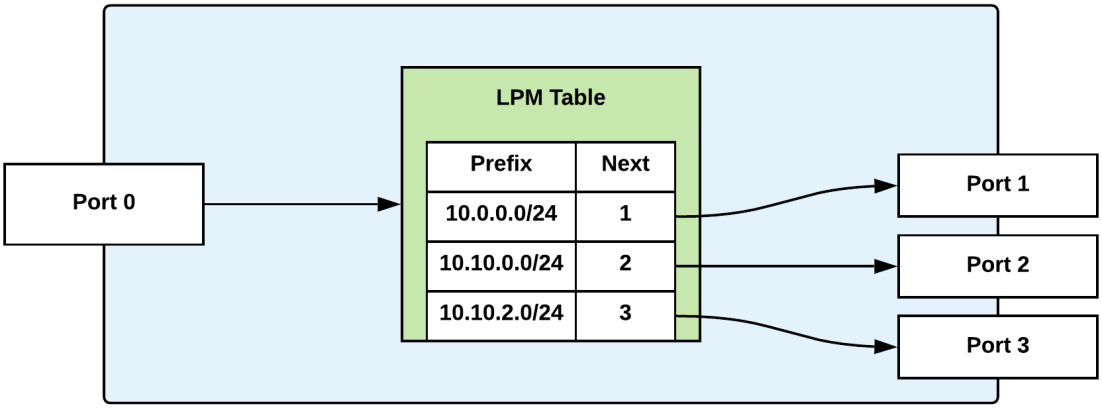


Figure 4.3: Shows a basic layer 3 routing pipeline.

Within the P4HC it is required to add a cache table into this pipeline. The cache table is smaller than the LPM table, but significantly faster. Therefore, it should be ideally placed before the main routing table. This design can be examined in Figure 4.4.

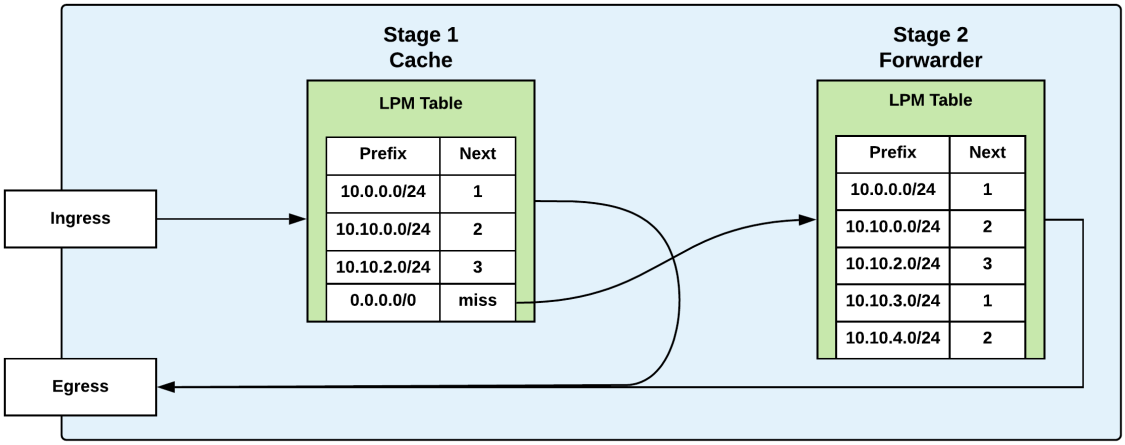


Figure 4.4: Shows a two stage layer 3 routing pipline with a cache and a forwarder.

Here both tables can bee seen. The first stage is the cache and the second stage is the forwarder. The cache only contains some prefixes which are heavily used. In contrast, the forwarder contains all available routes, so routes that are present in the cache are still present in the forwarder. However, the cache table requires an extra action on a cache miss. A cache miss occurs every time when there is no entry in the cache for a given packet. To perform a valid routing decision, the packet must then be sent to the forwarder table. To implement this behavior, it is possible to use a default action in this table or a default route. In this example the default route 0.0.0.0/0 is used, which only matches if no other table entry matches the current packet. The default route points to the forwarder table which should be able to perform a lookup. For example, a packet with destination address 10.10.4.1 is received at the ingress port. The Figure has only one ingress and egress port for simplifcation reasons. At first, the packet is matched against the cache. In the cache the only matching prefix is the default route, which means that a cache miss occurs. Now the packet must be sent to stage two the forwarder. The forwarder performs a lookup which results in a match, since it contains all available routes. The match is “Prefix: 10.10.4.0/24 Next: 2” which indicates that the packet should be sent out through egress port 2. In the figure port 2

is hidden behind the egress abstraction. This example should clarify the path of a packet in a cache miss scenario. Now a second example is provided which handles the cache hit case. So, a packet with the destination address 10.0.0.1 is received at the ingress. The packet is now matched against the cache in stage one. This results in a match on the entry “Prefix: 10.0.0.0/24 Next: 1”. The packet is then sent out trough port one through the egress.

These two examples show that in a cache-based pipeline there are two cases which lead to different packet paths in the pipeline: the cache hit case where the packets can be routed like in a traditional routing case and the cache miss, where packets must be matched again in the second stage. The provided pipeline should be sufficient to enable layer 3 routing with a cache stage.

However, the provided design has both stages in one device now. The main idea of P4HC was to utilize different devices for the cache and the forwarder. So, the provided pipeline must be split in two seperated data planes. In Figure 4.5 the pipeline was cut, and each stage was placed onto a different device. The devices are then connected to allow the transmission of packets between both stages. The in the previous example simplified physical ports are now re-introduced. To reduce the complexity of our example, packets only arrive at port 0 through the ingress and can only be sent through port 1 to 3 trough the egress.

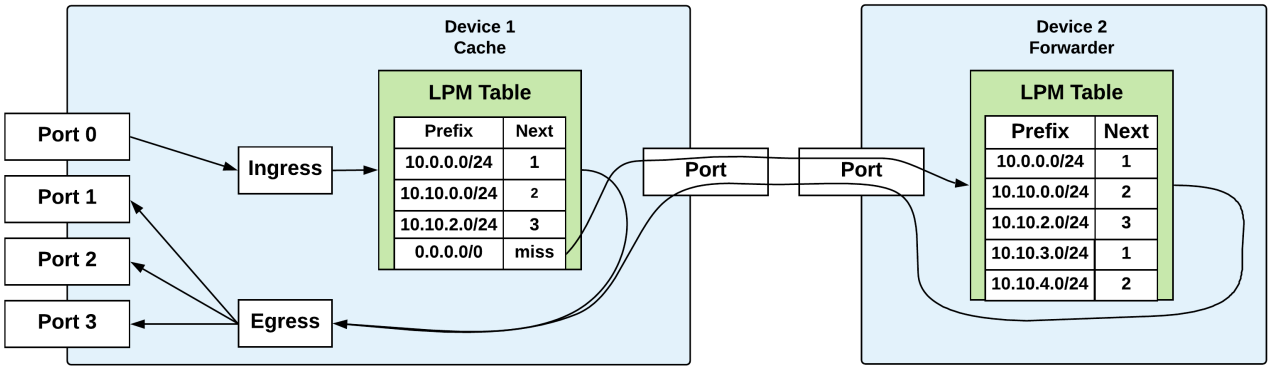


Figure 4.5: Shows the layer 3 routing pipeline with cache distributed to two devices.

Since the pipeline is now distributed over two devices, it is required to examine both cases, cache hit and miss, in order to evaluate if the pipeline is working correctly. The first case is the cache hit case. No additional evaluation is required, since the full packet path is on one single device as before. The interesting case is the cache miss scenario, where packets need to traverse both devices. To investigate this case, an example packet will be used. The packet with the destination address 10.10.4.1 is received on port 0 at device 1. Then a lookup in the cache is performed which results in a miss. Now the packet must be forwarded to the forwarder to make a routing decision. The forwarder is located on the second device, so the packet must be sent through the interconnect between both devices. Once the packet arrives at device 2, a lookup in the forwarder table is performed. The lookup results in a hit “Prefix: 10.10.4.0/24 Next: 2”. This is the point where problems with the naive splitting of the pipeline arise. The result of the lookup shows that the packet should be sent out though port 2. However, port 2 is located on the device 1, so the packet must be sent back through the interconnect. At this point, the problem clarifies: if the packet is sent back to the first device, device 1 has still no information on which port it should send the packet out. So, device 1 would still not be able to forward the packet and would drop it instead.

The learning of this observation is that both devices need a way to communicate in order to enable port awareness over both devices. There must be a way to tag the port information to a packet. So, a method is required to write port ids onto packets. A good solution would be to add a new header to a packet which contains the port information. The header should also not modify the packet on layer 3 or above, since we cannot risk any modifications at higher layers. Also, the new header should be able to

be added and removed seamlessly. A suitable header would be the vlan header which is placed at OSI layer 2 and is also standardized in the IEEE 802.1Q norm. The vlan header can be added and removed transparently and does not modify the upper layers. Also, the vlan header is heavily used in today's networks, so it should be supported on pretty much any platform.

So, every time a packet needs to travel between both devices the physical port information should be added to the packet through a vlan header. That means on a cache miss a vlan header needs to be added to the packet with the ingress port. Then the packet can be sent out to the forwarder. On the forwarder on a successful lookup, the next hop id which is the egress port id needs to be written into the vlan header. So, the vlan header must be modified. If the packet is back at device 1, the header must be removed and the packet has to be sent out through the egress port that is parsed from the modified header.

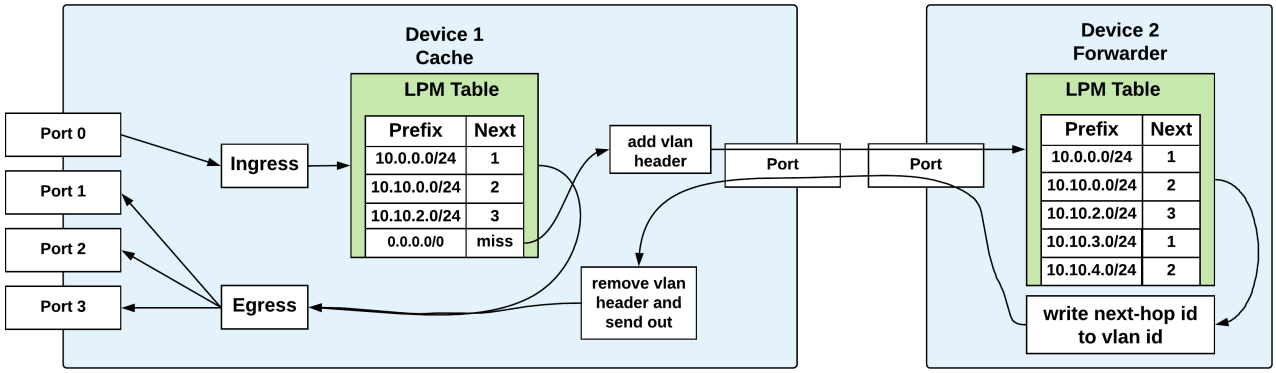


Figure 4.6: Shows how cache and forwarder are staged to enable data plane caching.

The introduction of a vlan header can be seen in Figure 4.6. To demonstrate the usage of the vlan header, the example with the destination address 10.10.4.1 is used again. The packet is received at port 0 and is matched against the cache. It only hits the default route which means a cache miss. The packet needs to be sent over to the forwarder. This time a vlan tag with the port id 0 is added to the packet before transmission. The modified packet then arrives on the device 2 and is matched against the forwarder table. The result is "Prefix: 10.10.4.0/24 Next: 2". Now the next-hop id, which indicates the destination port is written into the vlan tag. The packet is sent back to the cache device. Here the vlan tag is parsed and removed. Based on the retrieved vlan id 2, the packet will now be sent out to the correct port 2, instead of being dropped.

With the introduction of the vlan header, it is now possible to handle the cache miss case properly. The described pipeline now enables to perform layer 3 routing with a cache over two devices. This described pipeline is adopted for the data plane path of P4HC. In the next section it will be introduced how the P4 language is utilized to implement this pipeline on a hardware cache as well as on a software forwarder.

4.2.2 L2 Compliance

In the previous section the layer 3 pipeline of P4HC was presented. To ensure proper implementation of this pipeline in networks, it is required that the pipeline implements the lower layer 2 of the OSI model too. In this section the required addition as well as modifications are presented that are required for a minimal layer 2 integration. In normal operation a router connects multiple networks together. Each network segment that is directly connected to that router has its own layer 2 segment, usually Ethernet. An example scenario is presented in Figure 4.7 .

In the Figure 4.7 a router is used to connect two IPv4 subnets that are located on their own layer 2 segments. Host A and B are in the first subnet and host C is connected to the second subnet. Host A

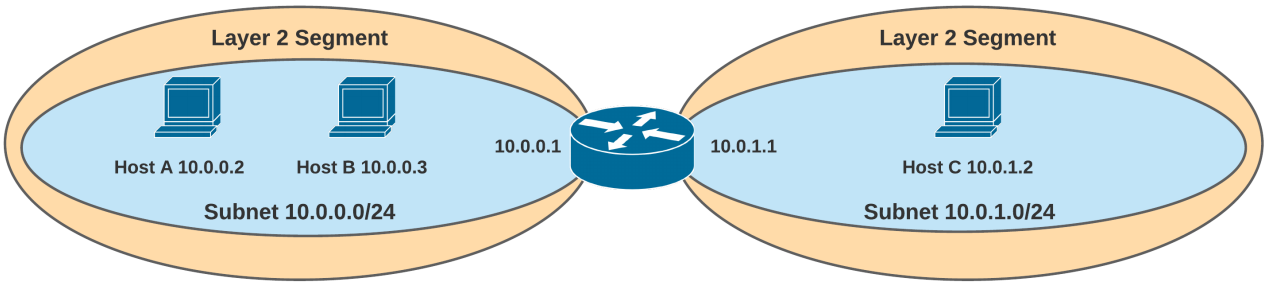


Figure 4.7: Two subnets in two different layer 2 Ethernet-segments connected by a router.

and Host B can directly exchange data without the requirement of a router, since they are located on the same subnet. If A wants to transmit data to host C, routing is required, because host A and C are located on different subnets with no direct layer 2 connectivity.

To establish the communication, host A must have the information that it is possible to reach host C via the router. This can be realized by a static routing entry on host A or by setting the router as the default gateway for host A. To send data to the router, the host must first resolve the router's layer 2 address which is represented by a MAC address. To resolve the router's MAC address, host A is using the Address Resolution Protocol [AB12]. Host A will broadcast an ARP-request to the layer 2 segment, essentially a message with the question where on the local network the IP address of the router is located at. If the router receives an ARP request on the network with an IP address that belongs to itself, it will answer the ARP request with its own MAC address. After host A has received an ARP response from the router, it can start transmitting data trough the router to host C. When the first data packets arrive at the router, the router now must determine where to forward the data to. First the router will perform a Longest Prefix Match on the layer 3 destination address to determine the target network. Afterwards the router needs to lookup the layer 2 address of the corresponding next hop. To lookup the address the router also will issue an ARP request to the target subnet to determine the layer 2 address of host C, since this host is directly connected and thereby functions as the next-hop. After the router has determined the layer 2 address of host C, it can now forward the data from host A to host C. When the router is forwarding the data between the subnets, it must rewrite the destination MAC address accordingly to the new target host. Also, when rewriting the destination MAC address, the router also rewrites the source MAC address to its own interface MAC address, so that the target host has the information to which host he can reply.

From the example above can be learned that it is important that our router must be able to handle the layer 2 of the OSI model as well. First the router must be able to respond to ARP Requests, second, it also must be able to send the packets to any host within a layer 2 segment. And finally, it must have the ability to rewrite the source MAC address of a packet on egress.

To implement ARP in our router, it is required to bind an IP address to an ingress port on our router. Based on the bound IP address the router can reply to ARP request for addresses that belongs to the configured IP subnet. For the identification and processing of relevant ARP request at the router, an ARP responder unit is required in the ingress pipeline. The ARP responder unit is directly placed after the ingress in the pipeline as shown in Figure 4.8.

The ARP responder unit consists of a table which contains an IP address, the ingress port and a corresponding MAC address. The IP address and the ingress port are then used to identify if an ARP request belongs to the router. After matching a relevant ARP request, the responder can generate an ARP response based on the corresponding MAC address from its table. To further clarify the ARP responder, an example is provided in Figure 4.9.

In the Figure 4.9 an ARP request for the IP 10.0.0.1 is sent out from host A and is received at port 1 of the router. The IP address as well as the port id is now matched against the ARP responder table. In this example the table has a corresponding entry with a MAC address. Based on this MAC address the

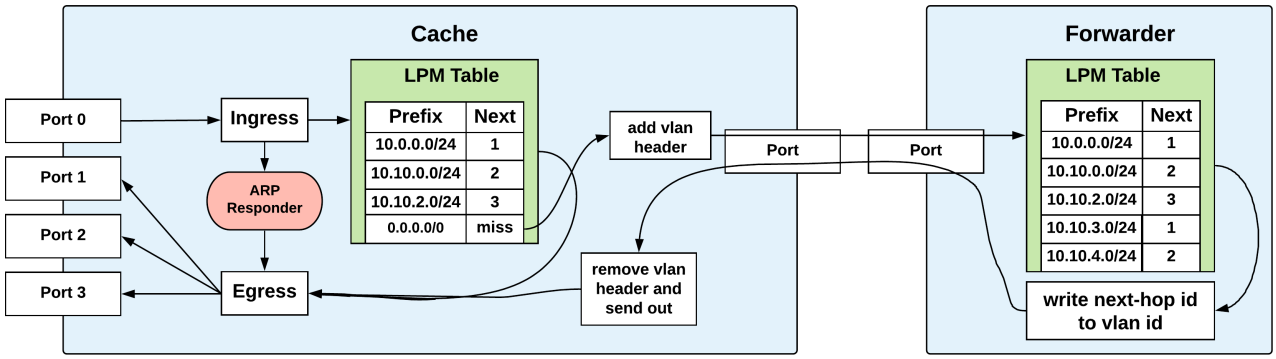


Figure 4.8: Shows the location of the ARP responder within the pipeline.

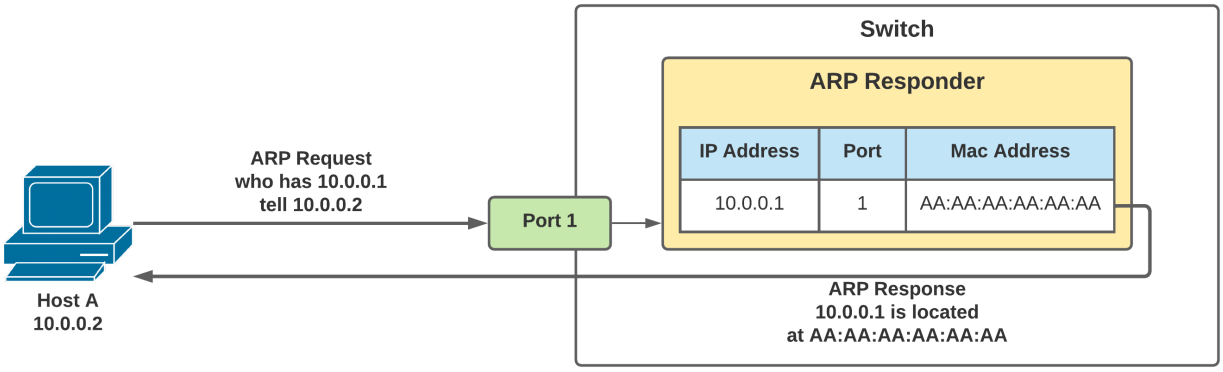


Figure 4.9: Example of how the ARP responder unit works.

ARP responder will generate an ARP response packet with the information that 10.0.0.1 is located at the corresponding MAC address and send the packet back to host A.

Now our pipeline can handle incoming ARP requests. Given that, the pipeline must also have the capability to rewrite the destination MAC address of a packet, based on the target host address. In this case it is not required for the pipeline to generate outgoing ARP requests, since we assume that the control plane is handling the discovery of the next hops' ARP addresses. A first naive approach would be to rewrite the destination MAC address based on the egress port. This would however, lead to the issue that it is not possible to have more that one next hop mapped to one egress port. To solve this issue, rewriting of the MAC address per next hop is required. The best solution for this task would be to add an extra field to the layer 3 LPM table in order to store the destination MAC address. This solution allows to rewrite the destination MAC address after the lookup when the next hop is determined. This allows to reduce the computational cost of an additional lookup within an extra table. The extension of the LPM table is presented in Figure 4.10.

Important is that the addition of the LPM table is required at the cache's and forwarder's LPM table. This requirement is based on the problem of the missing next-hop information at the cache when a packet is received from the forwarder which was described in the section before. Sure it would be sufficient to rewrite the destination MAC address upon removing the vlan tag on the cache, but this would require to add another stage within the pipeline, which can be prevented by adding the destination MAC address directly to the LPM table on the forwarder. The elegant design of using the vlan tag id equal to the egress port id - and thus preventing another table lookup - would not work anymore, since a port could have two next hops behind it.

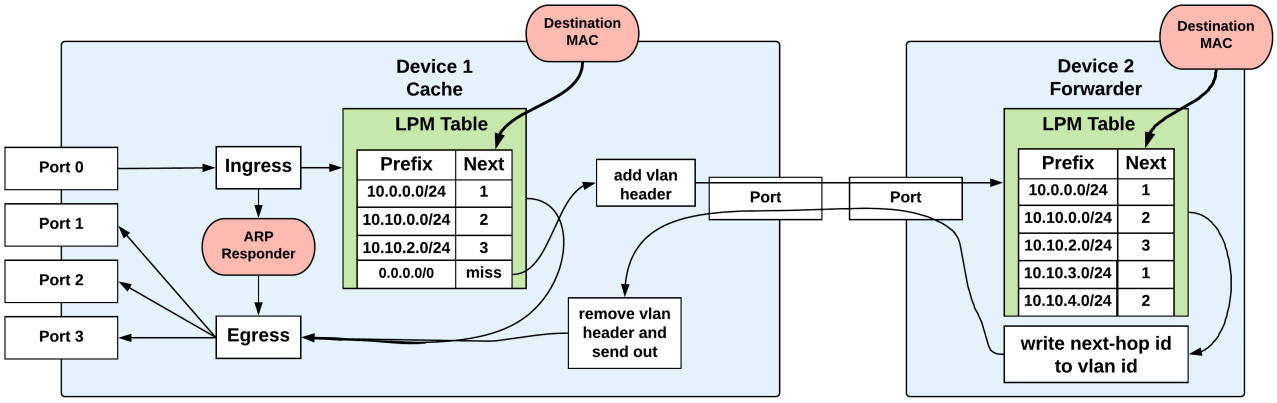


Figure 4.10: Additional destination MAC field in the LPM tables.

Finally, it is required to rewrite the source MAC address of a packet when it leaves the router. The rewrite of the source MAC address can be performed at the end of the pipeline on a per-port base, since one port could only have one MAC address. So, a source MAC rewrite unit is placed before the egress. The unit will have a table with the egress port-id and a corresponding MAC address. The placement can be seen in Figure 4.11.

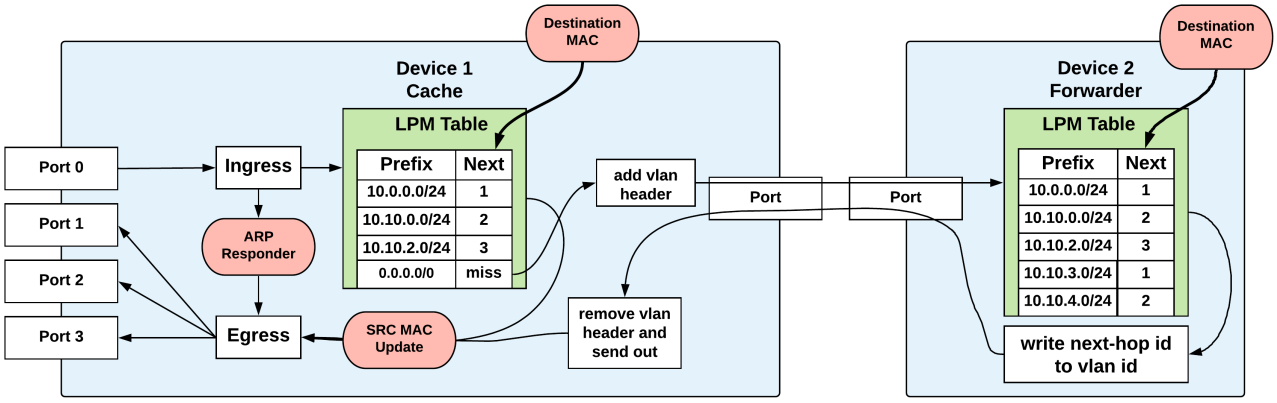


Figure 4.11: Source MAC address rewrite on egress.

The three presented modifications to the pipeline ensure the layer 2 compliance of our router. This compliance enables the P4HC pipeline to be able to integrate into any Ethernet-based network.

4.2.3 Metrics

As presented before, the goal of P4HC is to enable dynamic offloading between the cache and the forwarder. To perform the offloading, metrics of the traffic within the data plane are required. In previous work, the extraction of metrics was mostly realized by counter polling of the hardware device. The drawbacks of this approach were that it simply took too long to read out all device counters. Also, the read-out process could block resources on the hardware interface that could be used to update the data plane forwarding rules.

In order to realize a metric collection system without any interaction of hardware counters, P4HC will utilize a sampling approach. The pipeline on the cache will be extended in order to sample every n-th packet. The sampled packets will then sent out to a external collector, which is located at the controller. The sampling can be performed in the hardware of the cache without any interaction, which results in a great performance, since the whole management bandwidth of the hardware device is now available for routing updates.

After the sampled packets are sent out of the data plane, they will be received at a flow collector which will be part of the controller. The controller can now compute any metrics without consuming any hardware resources at the data plane.

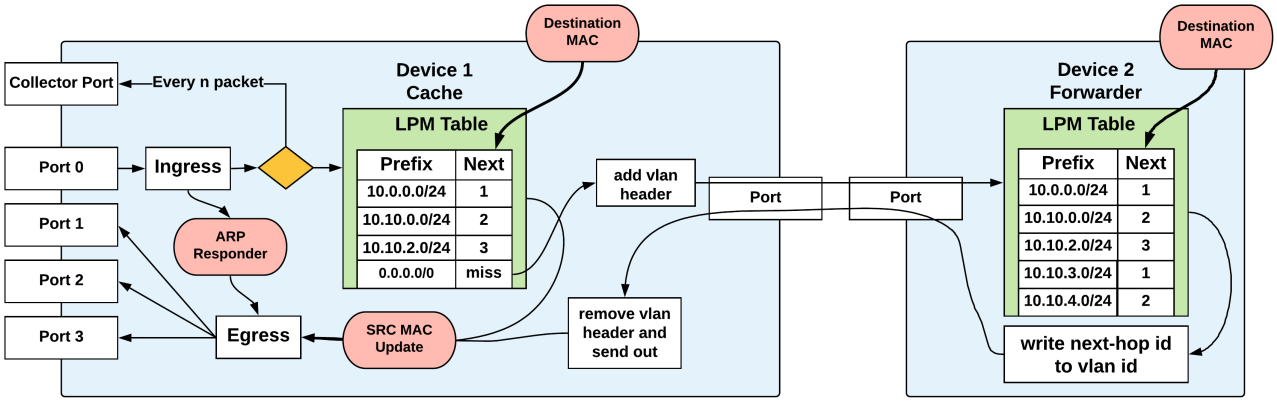


Figure 4.12: Sampling of packets on ingress of the pipeline.

In Figure 4.12 the placement of the sampler is shown. Important is that the sample unit only samples packets from externally connected devices. If we would sample all ingress ports, also the traffic between the forwarder an cache would be sampled, which would lead to incorrect measurements due to duplicate samples.

4.2.4 Splitted Dataplane with common definition language

In the previous section the routing pipeline for P4HC was developed. The developed design shows that there are two independent parts of the pipeline that must be implemented on different devices. To gain more simplicity and uniformity, both pipelines should be described in the same high-level language. Furthermore, it would be of benefit if the language was chosen to be platform-independent, so that P4HC could run on many devices.

All those requirements can be realized by the P4 Language. P4 provides an open language which enables to define custom switch pipelines. Also, P4 is platform independent since it operates on a logical higher level. The P4 Language describes a pipeline in a generic table, match-action fashion. In P4, tables can be used to match on any part of a packet and perform an action on a matched packet. To realize complex pipelines, multiple tables can be staged behind each other. Finally, the packet flow between

tables can also be influenced. To support multiple target platforms, the P4 Language comes with a toolchain that enables the usage of multiple back-end compilers. So, the P4 Language can be compiled to any target platform that has a backend-compiler available. Another advantage of utilizing the P4 Language for P4HC would be a common API to both P4 programs on the cache and forwarder device.

To implement P4HC in the P4 Language, two P4 programs would be required. One for the cache and one for the forwarder. Since the P4 Language is platform independent, every target platform for the cache and forwarder could be used. This enables P4HC to be fully platform independent while preserving maximum programmability. Additionally, through the selection of the P4 Language, P4HC will be fully protocol independent as well. In this work P4HC will be realized on a hardware switch and a software switch. The hardware switch will implement the cache and the software switch the forwarder. In Figure 4.13 the general idea of utilizing P4 for P4HC is presented.

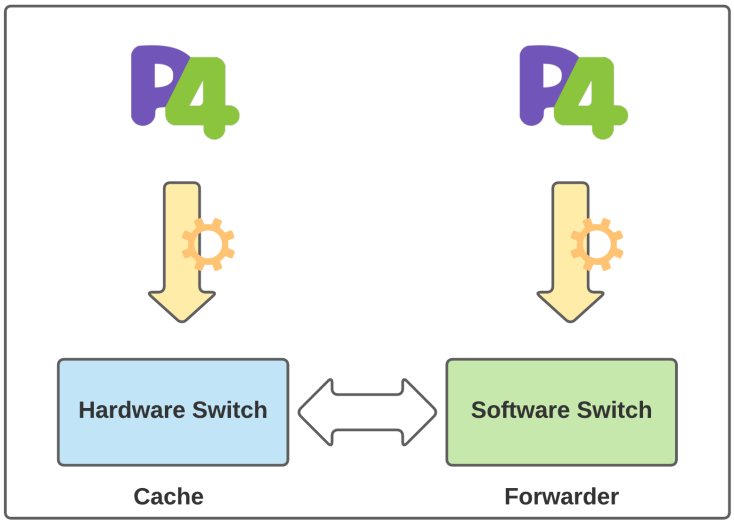


Figure 4.13: Utilization of the P4 Language for the cache and forwarder.

Since the P4 Language is relatively new, the variance of available P4 capable switches is still limited, the most popular one was selected: a switch with a Tofino ASIC from Barefoot Networks. The Tofino ASIC was the first ASIC which enables operators to fully design their own custom switch pipeline by utilizing a Protocol Independent Switch Architecture. Since the ASIC was built on purpose to support P4, it is the perfect candidate for the cache part at the time of writing.

The selection of a hardware platform for the cache was straight forward. Now a solution must be found to enable to execute a P4 Program on a common x86 computer. The simplest way to run P4 in software is the behavior model (bmv2) [bmv]. The bmv2 model allows to simulate a P4 program in software to enable rapid development in P4. However, bmv2 is more built up like a simulator, so any productive usage is not recommended and poor performance can be expected. As mentioned before in this work, to gain real line-rate packet forwarding in software, frameworks like DPDK are required. The consequence is that a P4 compiler back-end is required that allows to compile a P4 program to a DPDK application or a format that is executable by DPDK.

The idea of compiling P4 to DPDK can be found in Figure 4.14. The execution of P4 in DPDK should enable line rate performance at our forwarder. The documentation of P4 shows that there are different solutions to compile P4 to a DPDK executable. Since all those approaches are relatively new, we have selected the two most promising ones and have tested whether they are working as expected before we rely on them. For a simple proof of concept, the following two projects were selected: The official p4c-ubpf [p4c] back-end compiler in combination with p4rt-ovs [OTCK20, p4ra] and T4P4S (A Target-independent Compiler for Protocol-independent Packet Processors) [VHK+18].

Figure 4.14: Compiling a P4 programm to an executable DPDK application for a x86 server.

The p4c-ubpf [p4c] backend compiler is part of the official p4c compiler. The back-end translates a P4 program to user space BPF code. The Berkeley Packet Filter [MJ93], in short BPF, is a technology to process network packets on UNIX based systems. The BPF code describes a program that can be executed within the kernel at the lowest level or in a user space VM [ubp]. Also, it is possible to execute the BPF code in DPDK. In the case of the p4c-ubpf compiler, it is recommended to use p4rt-ovs [OTCK20, p4ra]. P4rt-ovs is an extension to the Open vSwitch (OvS). OvS is one of the most popular software switches on Linux based system and can run its data path in DPDK. The p4rt-ovs extension to Ovs enables OvS to execute BPF code in the OvS. This allows one to run a P4 program in an Open vSwitch that uses DPDK for line rate packet forwarding. Unfortunately, we were not able to successfully compile a P4 program with the compiler, neither it has worked to execute the uBPF code on p4rt-ovs.

The second approach to compile a P4 program to DPDK is T4P4S (A Target-independent Compiler for Protocol-independent Packet Processors)[VHK+18]. T4P4S is a separate compiler that allows to compile P4 to code against a networking hardware abstraction layer (NetHAL). This generated NetHAL interface can then be implemented on different targets, like using DPDK for example. In our test we could successfully compile a P4 program to a DPDK executable. However, we were not able to send packets through the application. Furthermore, T4P4S provides no external common control plane API, so we would have to implement this by ourselves anyway.

The PoC shows that there are promising approaches to compile a P4 program to DPDK, but the tools and software are still not ready at the moment. To overcome this lack of available software, a substitution for the P4 program at the forwarder is required. As DPDK is the preferred target for a potential P4-to-software-switch compiler, the decision was made to implement the forwarder directly using the DPDK framework. This decision was based on the following reasons: first, the performance of a plain DPDK implementation should be comparable to a P4 to DPDK compiled program, as the forwarding technology is the same. Second, DPDK itself comes with the libraries that would be required, like for example for performing LPM lookups. Finally, the DPDK application can be implemented as a closed component with a common external API that allows to substitute the plain DPDK application with a P4 one, once the tools are ready.

To summarize, the data plane of P4HC will be implemented as following: an P4 program that is executed on a hardware switch as cache and a plain DPDK program as the forwarder on a common x86 server. The setup is finally presented in Figure 4.15.

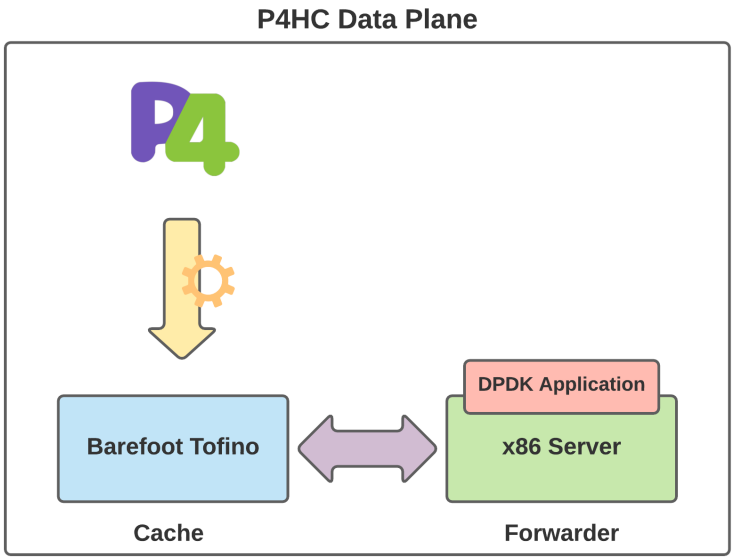


Figure 4.15: Technology stack for the P4HC Data Plane implementation.

4.3 Control Plane

In the previous section, the data plane of P4HC was introduced. In this section the controller design of P4HC is presented and examined. First, it will be introduced which tasks belong to the controller, secondly each task will be discussed in more depth.

Within the controller the complete control logic of P4HC is realized. At first the controller must be able to configure the data plane. For this essential task, two interfaces are required, one for the cache and one for the forwarder. For proper functionality, the controller must first initialize the data plane and then feed the routes into the data planes. In order to install the routes, the controller must collect the route information from its neighbour routers through a routing protocol. Afterwards it must store the route information within a RIB for future processing. When the routes are stored in the RIB, the controller must then install those routes into the forwarder. When installing the routes, it must be assured that all requirements are initialized correctly for the next hops of the given route. In order to make an offloading decision, the controller must also collect the metrics for each route to determine how much traffic per prefix is traversing through the data plane. Based on this information, an offloader component within the controller can decide which prefix should be installed into the cache to reduce the load to the forwarder. When the controller has decided the prefixes that should be installed into the cache, it must ensure that the forwarding configuration is still valid afterwards. In order to guarantee the preserving of the forwarding behavior, the controller will have a dependency resolver component, which will identify problematic prefixes and deal with those dependencies. Finally, the controller must implement a cache replacement strategy. If a new route needs to be installed into the cache while the cache is full, the controller must remove some cached prefixes. The identification and revocation of those routes must also be coordinated to preserve the correct forwarding behavior. All those parts and components can be seen in Figure 4.16. Here all major components of the controller are presented. Each component will be examined and discussed in the upcoming sections.

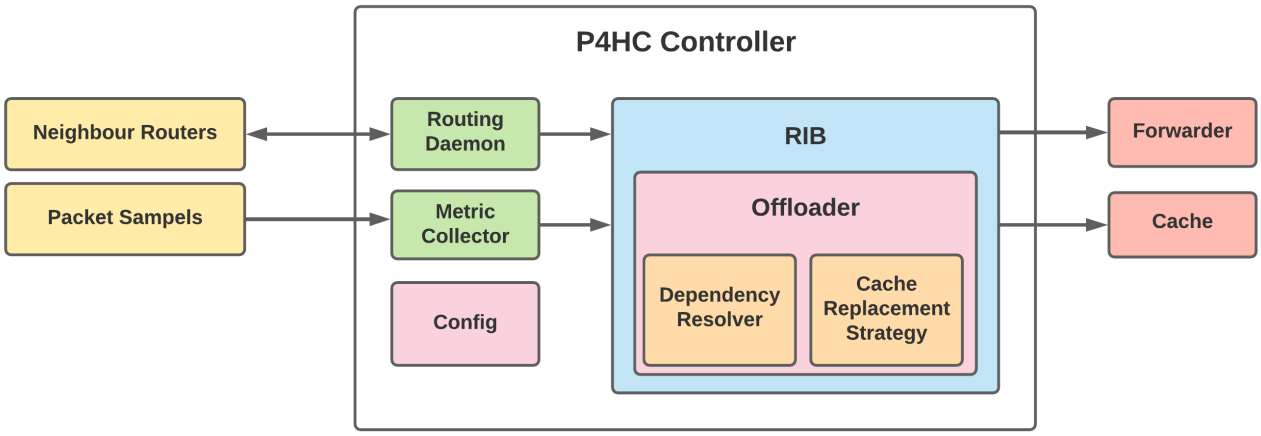


Figure 4.16: Overview over the main components of the P4HC controller.

4.3.1 Interfaces

The controller has three major external interfaces. The first one is the north-bound interface which enables the outside communication of P4HC. The interface will implement the routing protocol to exchange data with the neighbour routers. The remaining two interfaces are the data plane interfaces which are the south-bound ones. These two interfaces realize the interaction between the forwarder and the cache. At first the routing interface is presented and afterwards the data plane interfaces.

The north-bound routing interface has the main task of exchanging the routing information with the neighbour routers. As mentioned in the introduction, the most common protocol to exchange routes between routers is BGP. Based on this reason P4HC will leverage BGP as routing protocol as well. The BGP protocol is well defined in RFC4271 [RHL06], so at this point the protocol will not be discussed in detail. Even if BGP is the most frequently used option, there are still many other routing protocols available. To enable the integration of those later on, the P4HC controller will abstract the routing protocol in a routing daemon component. The routing daemon component will push the learned routes to the RIB. So later on, another protocol can be implemented in the routing daemon which then pushes the routes to the RIB. A block schematic can be seen in Figure 4.17.

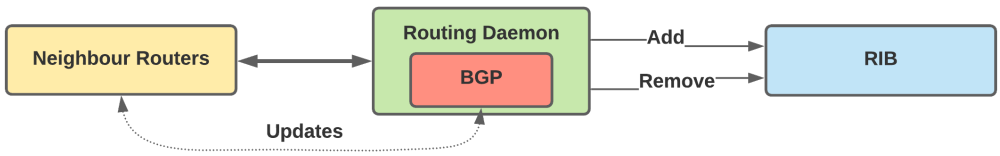


Figure 4.17: The routing daemon pushes all routes to the RIB.

After the south-bound interface is completed, the two remaining north-bound interfaces must be defined. The main task of those interfaces is to insert routes into the data plane. We decided that both interfaces should utilize the same type of API technology. A common API technology enables rapid development while also reducing the amount of dependencies in the controller. Another reason is that the long term goal of P4HC is to support P4 enabled data planes on the cache and forwarder. If both data planes utilize the same technology, the API technology should also be the same to simplify the design. However, even if the API technology may be the same at the cache and the forwarder, both APIs have different methods due to the difference in their data paths.

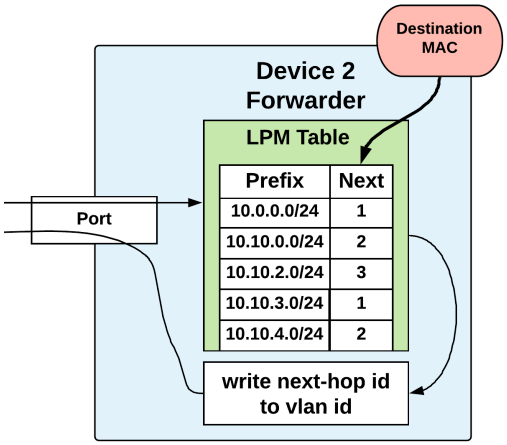


Figure 4.18: The forwarders data plane.

At first the forwarder API will be defined. In Figure 4.18 the data plane of the forwarder can be examined. The forwarder data plane only consists out of one longest prefix match table, so only two methods will be required. One for installing new routes and one for removing existing ones. To add a new route, the following parameters are required: first the route itself as prefix. Second the egress port which will be represented by the vlan tag, the next-hop MAC address and finally the next-hop IP address. The next-hop IP address itself is not necessary for the forwarding itself, but can be used as an index for the next-hop within the data plane, if required. The last API method will be the route revocation, which allows to remove a route from the forwarder. In this method only the route itself will be required as parameter. So, the forwarder API has two methods which are presented in Figure 4.19.

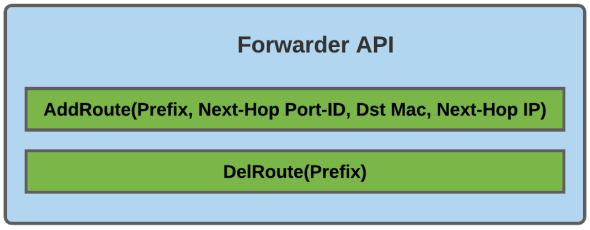


Figure 4.19: The API methods of the forwarder interface.

After the forwarder's API methods are defined, the cache methods need to be defined too. To extract the required methods, Figure 4.20 shows the data plane pipeline for the cache. As well as the forwarder, the cache also has a LPM table for the routing. To install and remove routes, there also will be two methods, one for installing new routes into the cache and one for removing the routes. As seen before, the addition of routes requires the following parameters: the route itself the next-hop port and the destination MAC address. For the deletion of the route, again only the route as parameter is sufficient.

As shown in the pipeline, the cache has additional components which require additional methods compared to the forwarder. At first the cache has an ARP responder that will also bind an IP address to a port of the cache. In order to bind an IP address to the port, the IP address itself is required, the port id and the corresponding MAC address of the port. To remove an IP address from the port, the IP address and the port id are required. The port-id itself is not sufficient, because it is possible to bind more than one IP address to one port of the cache.

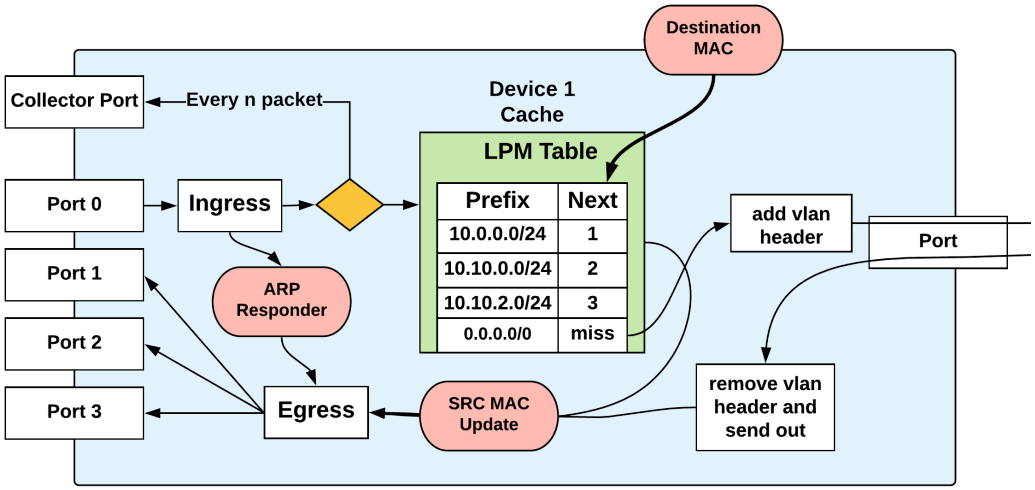


Figure 4.20: The cache's data plane.

Also the pipeline has the capability to rewrite the source MAC address, based on the egress port. To enable the source MAC rewriting, a method is required with the MAC address and the port-id as parameters. To remove the source MAC address from the port, only the port-id is required, since only one MAC address can be added to one port. Finally, there are two methods required for the setup of the cache's data plane. One method to define the port of the forwarder and one for defining the port through which the sampled packets are sent out. All methods of the cache's API can be examined in Figure 4.21.

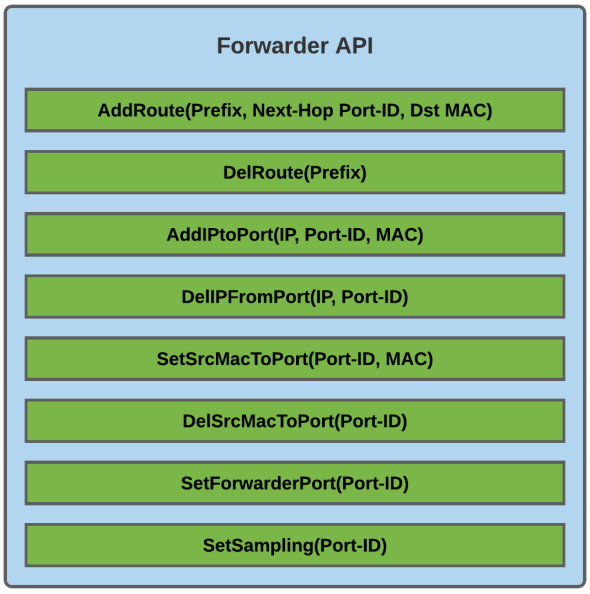


Figure 4.21: The API methods of the cache interface.

4.3.2 Routing Information Base

The central data storage of the controller is the Routing Information Base. The RIB will store all available routes of the router. Also, the RIB needs to coordinate the installation of those routes to the data plane.

In most cases the RIB can contain multiple routes per destination networks. In this case, only the best-matching route will be installed into the Forwarding Information Base (FIB), which is then mirrored to the data plane. In the case of P4HC, the RIB will be equal to the FIB, since the main goal is to prove that the general data plane concept is feasible. A real dedicated RIB and FIB can be integrated later.

To store all routes the RIB needs a data structure which can store additional information together with a route. The most important information to be added is the next-hop, when a route is installed to the forwarder and when a route is installed into the cache. The next hop field contains the information to which port the packets need to traverse, as well as the information which destination MAC address the packets must be sent to. Since many routes share the same logical next-hop, it would make no sense to store all information together with every route. To reduce the storage of redundant information, a next-hop lookup table is introduced. The next-hop table will contain all available next-hops, so a RIB entry must only refer to a next-hop entry from this additional table. As an index for this table, the next-hop's IPv4 address would be sufficient, since it should be unique. This principle is provided in Figure 4.22.

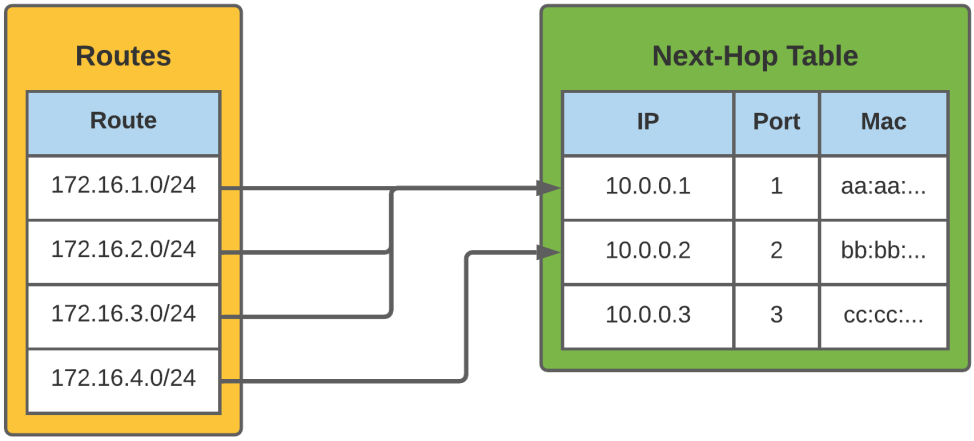


Figure 4.22: Next-Hop table is used to minimize the amount of redundant data in the routing table.

Here can be seen that the first three routes share the same logical next-hop, so the next-hop information is stored only once. Those savings become more relevant if a full table is installed into the RIB. An easy example would be an edge router within a provider network, which is connected to five other networks. In this example there would be only five available next hops. Through the addition of the next-hop table, we can reduce the number of stored next-hop data by around 849.995, given that we have our five next-hops and a full table size of 850.000 [cid].

After the next-hop information for the RIB is stored efficiently, the routes need to be stored efficiently as well. In this case efficiency can't be seen as saving memory for storing the routes, since there is no route that can be optimized out. Efficiency for the routing information is more important at the lookup side. For example, the utilization of a simple list or array to store all available routes would not be sufficient, because lookups would require to search through every entry. This search would result in a linear runtime, which is still not fast enough. Also searching more specific prefixes would require iterating through all routes. A more sufficient way is to store the routes within a Radix tree, which is a space optimized binary tree. The basic idea is to build up a binary tree out of the prefixes' addresses. Since an IPv4 address consists of 32 bits, the maximum possible depth of the tree would be 32. So in comparison to a list or array, we can find an entry within a maximum of 32 lookups instead of n. Since the radix tree is space optimized, it will also eliminate nodes and edges that store the same information, so the depth of the tree will be further minimized. Also, a prefix tree allows for an efficient LPM. To find a longest prefix, the tree must only be traversed until the last node or leaf that still matches the address is reached. The resulting node will be the longest available prefix with a maximum of 32 lookups.

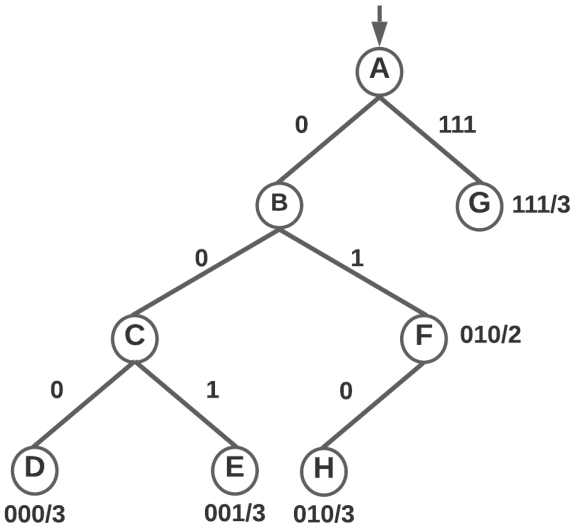


Figure 4.23: Example of a Radix prefix tree which stores all prefixes.

In Figure 4.23 an example prefix tree is provided with an address space of 3 bits. To clarify the function of the prefix tree, some examples are provided. The first one would be the lookup of the address 001. To check if the tree contains the address, the lookup is started at the root node A. The first bit of 001 is 0, so the lookup will continue at node B. The second bit is still a 0, so the lookup continues to node C. The last bit is a 1, so the lookup will continue to node E. Node E is a leaf, which means no further descending is possible. Since node E also equals the address that needed to be looked up, it can be confirmed that the tree contains the given address. The second example would be the lookup of 111. Here the compacting of an radix tree can be seen. The lookup starts again at the root node A. The first bit is a 1, so the lookup continues to the right side of the tree. The only available node is a leaf storing 111, so the lookup will continue at node G. Since there are no other options available, the tree has compacted the additional lookup into one leaf. Since node G is a leaf, the requested address is compared with the address of node G resulting in a match and thereby confirm that the tree also contains 111.

The final example is a longest prefix match. So, the example question is which is the longest prefix for 011. The lookup also starts at the root node A. Then the first bit is compared, which is 0 and points to node B. At node B, the second bit is compared which is a 1 and points to node F. At node F there is no successor available for the last bit being set to 1. This means at node F the longest possible prefix for address 011 is found.

The Radix tree allows to store all routes within the RIB in an efficient way, while also guaranteeing fast lookups. So if the routes from the routing daemon are received at the RIB, they will be installed to the prefix tree and assigned a reference from the prefix to the corresponding next-hop entry from the next-hop table.

Afterwards, the RIB must install the newly learned routes into the forwarders data plane. As described in the previous section, the forwarder must contain all available routes. So essentially the forwarders data plane consists of a full mirror of the RIB. Finally, the RIB must be able to install routes into the cache. To decide which routes need to be installed into the cache, the RIB will instantiate a new offloader component and provide an API method to the offloader, in order to receive a list of prefixes which should be installed into the cache. Once the RIB will receive the list of prefixes that should be installed into the cache, it must first check which routes are already present at the cache. For this task, the RIB will look up all prefixes internally that were installed into the cache before. Afterwards the RIB will compare the currently installed routes with the new list from the offloader. While comparing the two lists, the

RIB will create a list of prefixes that need to be removed and a list of prefixes that must be added to the cache. Important is that the RIB will first apply the remove list to the cache and afterwards the addition, to maintain the correct forwarding behavior. Also, it is important that both lists are sorted based on the prefixes' length. In the addition case it is important that the more specific prefixes are installed first. This solves the problem that traffic for the most specific prefix is forwarded incorrectly during the installation of a less specific route, which also matches that traffic. On the deletion it is important to remove less-specific prefixes first and the more specifics afterwards. This ensures that there is no temporary incorrect forwarding behavior to be expected. The offloading process is also summed up in Figure 4.24.

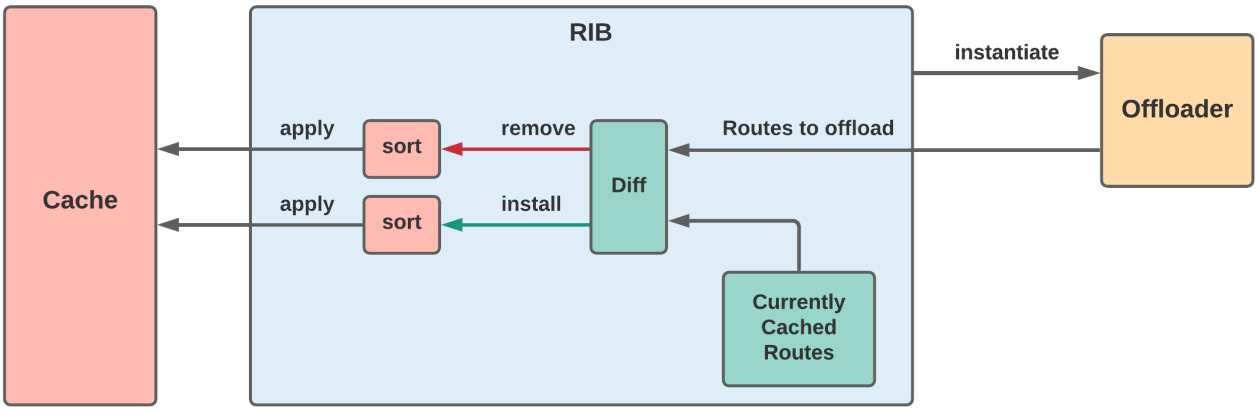


Figure 4.24: Workflow for offloading prefixes into the cache.

4.3.3 Metric collector

In the last section the offloader was introduced, but not described in more detail. Before the offloader is explained in detail, the metric collector will be introduced, since the offloader will rely on those metrics for making an offloading decision. As mentioned before, to decide which prefix must be offloaded, it is required to know which prefix is responsible for how much traffic. As described in the data plane chapter, those metrics need to be generated based on the sampled data that is created by the data plane. So, the task of the collector component is to compute the traffic per prefix out of the packet samples and then store them for the offloader.

At first the collector analyzes the sampled packets. In order to do so, the collector must receive the sampled packets and parse the following information: the destination address in order to aggregate the metrics per prefix later on and the size of the packet to calculate the amount of data per prefix.

After the packets are parsed, the data must be aggregated on a per prefix base. For this, the collector processes the complete routing table from the RIB. As a result, the collector can map the destination address of each packet to the corresponding prefix of the routing table based on a LPM lookup. Based on the prefix result, the collector will aggregate the metrics on a per prefix base. For each resulting prefix the collector will count the number of packets and the total received bytes per prefix, based on the packet length. To generate representative metrics as packets per second and bits per second on an per prefix base, the collector will read the collected data every n seconds and clear the current data from the aggregator. Afterwards, the collector must renormalize the data since the input was only sampled data. Also if the interval of the readout is lager than one second, the data must be renormalized to a per second metric. Once the data is finally normalized, it can be stored within a database for the offloader later on. Figure 4.25 shows the internal processing of the samples by the collector.

As described before, the metrics finally need to be stored within a data base. The simplest approach would be to store the last available set of metrics in a table. Then the offloader can access the last

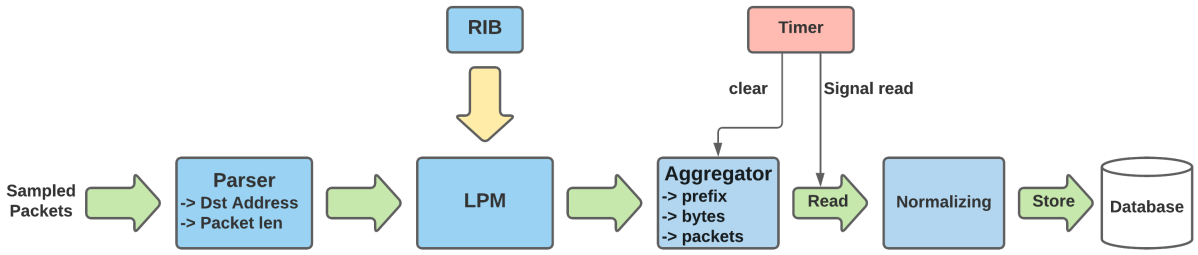


Figure 4.25: The metric collection pipeline of P4HC.

available metrics in order to make a decision. However, the offloader would lose the ability to query historical data for smarter decisions. Since the collector generates continuously data, it would be ideal to store the information within a time series database. Those databases allow to store a datapoint with a timestamp. With the storage of the metrics in a time series database, it is be possible to review the metrics over a time span. An example would be to the average traffic of one prefix over the last 5 minutes. From our perspective, this is the best approach to store the data, because it allows to review the networks' traffic over a longer time-span instead of just the information from the last seconds.

4.3.4 Offloading

The last missing component within the controller is the offloader itself. The offloader has the task to identify the routes that need to be installed into the cache, resolve the dependencies of those rules and implement a cache replacement strategy to be used when the cache fills up.

The first major task is to identify the prefixes that need to be cached. This selection process is a continuous process that must be repeated over time. This is quite important, because the traffic patterns of a network can and will change over time. An example is that over night there may be more backup traffic within a network that will disappear after a backup job is finished. Thus, it makes sense to cache the routes over the duration of the job and revoke them afterwards.

Another issue in the selection process would be patterns, like regular bursts. An example would be two servers within a network that synchronize data every minute for a duration of 10 seconds. If the offloading process would run every 10 seconds with the currently latest metrics, the route between those hosts would be cached and removed from cache continuously. The reason would be that the offloader would see the flow between those two servers for 10 seconds and when the offloader runs afterwards again, the flow would already be inactive. This would lead to the offloader removing the route from the cache again. After 50 seconds the flow will become active again which leads the offloader to reinstall the route into the cache. This cycle would repeat over and over, as shown in Figure 4.26.

These cycles must be avoided, because updates to the cache should be minimized. The issue is that updating the hardware cache comes with high cost and oftentimes the cache has a limited update rate. To finally solve this issue the offloader should be able to make use of historical data. As it was described in the metrics section, the collector will store the metrics in a time series database. This enables the offloader to view the volume of the traffic over time.

To make an offloading decision, we have decided that the offloader will query the last minute of available data in order to make its decision. To identify the most active prefixes over the last minute, the offloader will review the volume of traffic for every prefix over the last minute. Then it will select the n most active prefixes over the last minute. The selection of n here is also not trivial, because of the unknown cost in terms of dependencies at this point. The Problem that arises is that the cache has only a certain amount of space available. So a naive approach would be to select n to the number of available cache entry, the issue here is that there would be no space left in the cache to handle possible flow rule dependencies and thus to maintain the correct forwarding behavior. We concluded that there is

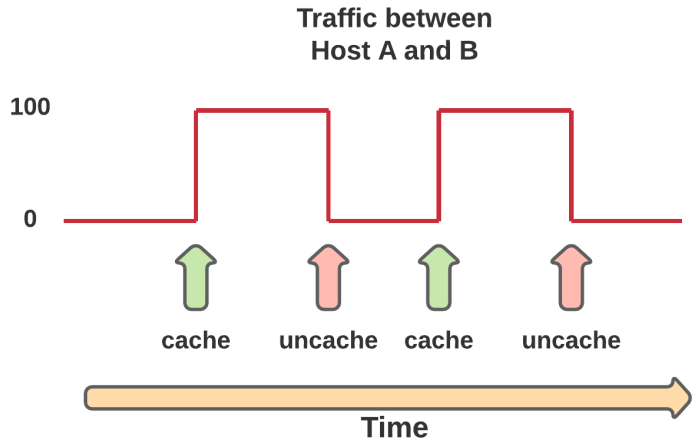


Figure 4.26: Hosts that communicate only every few seconds can create cache and uncache cycles.

no ideal size for the parameter n, because it can't be exactly predicted how expensive the installation of rules through dependencies would be for any situation. Also, if n is selected to a relatively small number, valuable cache-space is wasted for possible dependencies that may not even exist. So the naive approach with n equals cache size with a small extension might be a suitable option.

The solution we have chosen is to select n to be equal to the cache size. If the n most active prefixes are selected, the cost for each prefix in terms of dependencies is calculated. Afterwards the list of prefixes is sorted based on their traffic volume. The prefixes with the most traffic, including their dependencies, will be at the top since they are the most beneficial candidates for the cache. Afterwards, the list is simply cut off after the cache size is exceeded. The cost of a dependency can been seen as the number of rules that need to be installed additionally to the prefix, so the computation is simplified.

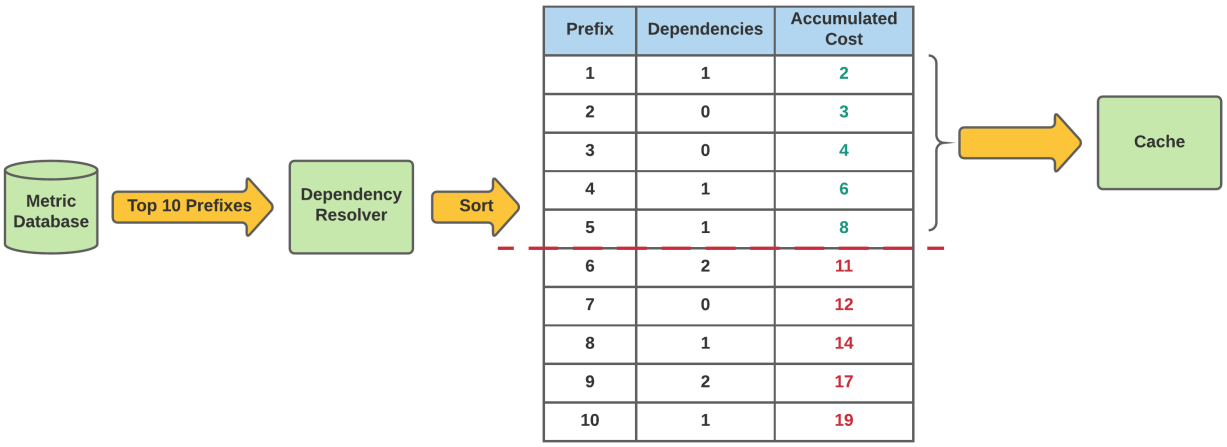


Figure 4.27: The offloading selection process with cache size equals 10.

An example of the selection process is provided in Figure 4.27. In the example the cache capacity is 10 entries. At the first step, the offloader will query the ten most active prefixes from the database. Afterwards it will calculate the dependencies of each rule. Then the list is sorted based on the traffic level of the original top ten prefixes. When calculating the total cost per prefix to each prefix in the list, the cost of the dependencies must be added. For rule 1 in the example the total cost is 2, because the prefix itself requires one entry and the one dependency also requires one additional entry. At the point

in the list where the cost is exceeding the cache size, a cut must be performed. The example shows that even with this approach a 100 percent cache utilization can't be achieved. The total cost until rule 5 is 8. The next rule has a total cost of 3 (1 for the prefix itself and 2 for the dependencies). So rule 6 can't be installed completely into the cache, since there is not enough space for all dependencies which would lead to an incorrect forwarding behavior. This means only rules 1 to 5 can be installed into the cache. The selected rules are then sent over to the RIB. The RIB will update the cache accordingly.

As the selection of the offloaded prefixes is now defined, it must be discussed how the dependencies for each prefix are generated. In the Related Work (Chapter 3) many approaches for resolving dependencies were provided. As this work will have a closer focus to data plane performance, we selected the strategy which is the easiest to implement. The strategy is to install all dependencies for a rule into the cache. This approach is also described in CacheFlow as Dependent-Set. This approach is clearly not ideal in terms of cache space utilization, but is sufficient for our case, since we will focus more on the data plane within this work.

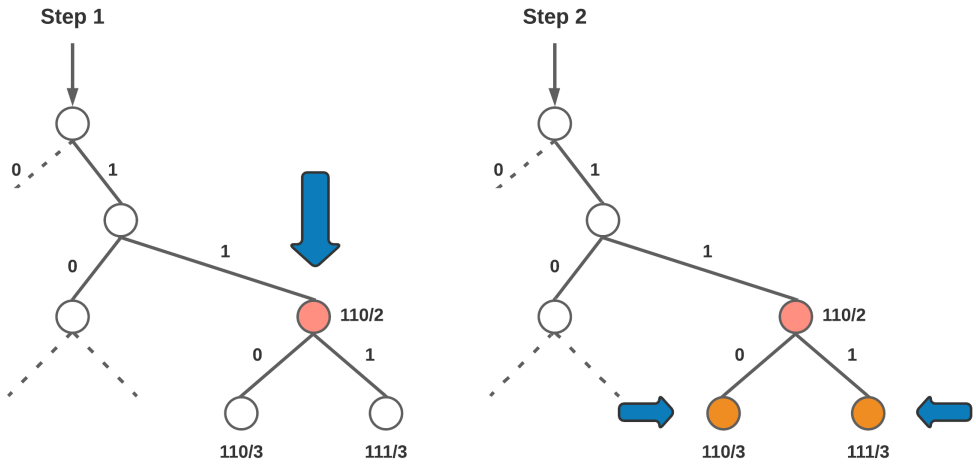


Figure 4.28: The two required steps to find all dependencies for prefix 110/2.

In order to generate a Dependent-Set the full RIB is required. As it was described before, the RIB stores all prefixes within a Radix Tree. To visualize how all dependencies can be identified within a Radix Tree, an example in Figure 4.28 is provided. Before the example is explained, it should be noticed that the dependencies of a prefix are all more specific prefixes within the RIB. The example uses a 3 bit addressing space and the prefix that should be offloaded is 110/2. So the goal is to identify all more specific prefixes within the tree. In Step 1 the prefix 110/2 is searched which is highlighted red. After the prefix is found, the more specific prefixed must be searched. Within a Radix tree all more specific prefixes are successors of the current node. So every node or leaf after the prefix 110/2 is a dependency. In this case the orange highlighted prefixes 110/3 and 111/3. This simple algorithm allows to identify all dependencies from a prefix with a minimum effort.

Finally, in classic caching scenario a cache replacement strategy is required. The strategy is applied once the cache is full and a new route needs to be installed. However, since our offloader generates the full cache set at every iteration, no explicit cache replacement strategy is required, since it was handled implicitly trough the selection of the n most active prefixes. This process can be compared to a Least Recent Used (LRU) cache replacement strategy. The LRU removes the entry that was used the least during a certain a amount of time. Since we select the n most active prefixes from the database, we implicitly remove the least recently active ones, if there is a new prefix that handles more traffic.

4.4 Summary

In this chapter the design of P4HC has been presented and discussed. Each component of P4HC was explained and described in detail. In the next chapter the implementation of the prototype of P4HC will be presented.

5 Implementation

In the previous section the design of P4HC was discussed. In the following section the implementation of the prototype of P4HC will be presented. Since P4HC consists out of multiple components, mainly the data plane, the metric collector and control plane, this chapter will be splitted in three sections: first the implementation of the data plane is provided, second the metric collector and finally the implementation of the control plane is explained.

5.1 P4HC Data Plane

In the next sections the implementation of the data plane is provided. The data plane of P4HC consists out of the cache and the forwarder. First, in Section 5.1.1 the implementation of the Tofino cache is presented. Second, in Section 5.1.2 the implementation of the DPDK software forewarder is presenetd.

5.1.1 P4HC Cache

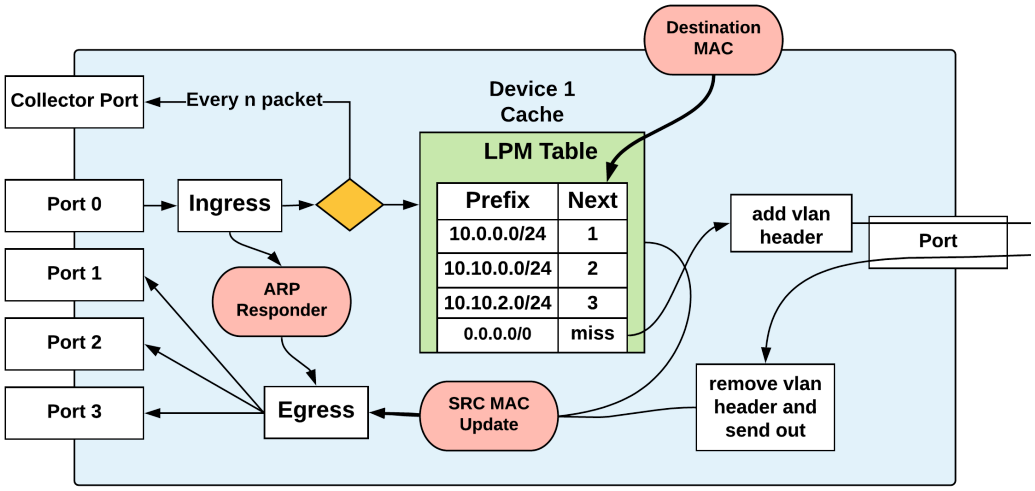


Figure 5.1: The cache's data plane pipeline.

As mentioned in the design, the data plane of the cache will be implemented using a Barefoot Tofino hardware switch (WEDGE 100BF-65X [wed]). To program the switch a P4 program must be implemented that represents the before designed pipeline of the cache. The cache's pipeline is also provided in Figure 5.1. To implement the P4 pipeline, the first step is to identify all headers that will be required within the pipeline. The required headers are: the Ethernet header, the ARP header, the vlan header and finally the IPv4 header. The Ethernet header is required in order to manipulate the source and destinations MAC addresses, the ARP header to bind a fictitious IP address to the switch, the vlan header to identify the egress port and the IPv4 header to perform routing.

In order to parse the headers in P4, the headers must be defined before. An example definition of the Ethernet headers is provided in Listing 5.1. The Ethernet header consists out of three fields. First the source and destination MAC addresses which are 48 bits long each and a type field with 16 bits that identifies the next header on the packet. It is important that the order of the header fields is the same as they appear in the packets, since the parser will read the headers in the defined order.

```
typedef bit<48> mac_addr_t;

header ethernet_h {
        mac_addr_t dst_addr;
        mac_addr_t src_addr;
        bit<16> ether_type;
}
```

Listing 5.1: Example definition of a header in P4.

After the headers are defined, the parser can be implemented. The parser will parse the defined headers from an incoming packet. As described before, the parser in P4 is represented by a parse graph. At each node a header can be extracted and afterwards a transition can be performed. Also, the transition can be performed conditionally, based on the parsed header values. An example node of the parse graph for the Ethernet header can be examined in Listing 5.2. In the example can be seen that at first the Ethernet header will be extracted from the packet and afterwards a conditional transition is performed to the next state. The condition is based on the Ethernet type, which describes the next header of the packet. Based on the type, the next state according to the header will be selected.

```
state parse_eth {
        pkt.extract(hdr.ethernet);
        transition select(hdr.ethernet.ether_type) {
                ETHERTYPE_ARP: parse_arp;
                ETHERTYPE_VLAN: parse_vlan;
                ETHERTYPE_IPV4: parse_ipv4;
                default: reject;
        }
}
```

Listing 5.2: Example node definition of the P4 parsing graph.

After the parser is defined, the pipeline itself must be implemented. The first step is to identify the required match action tables. From the pipeline design the following tables and tasks can be extracted: First the ARP responder requires a table that matches incoming ARP packets. To identify the relevant ARP requests, the table must match on the requested IPv4 address, the ARP OP code, the source MAC address and the ingress port with an exact match. To answer to matching ARP requests, an action must be defined that can then be executed. The definition of the table and the action can be examined in Listing 5.3.

The action to answer to an ARP requests requires a MAC address as parameter for the ARP response to be generated. To then generate the ARP response, the existing ARP packets is utilized and the header fields are rewritten to an ARP response. At first the egress port will be selected based on the ingress port in order to send the reply to the correct Ethernet segment. Afterwards, the MAC addresses of the Ethernet header will be set and finally the ARP header is modified to an ARP response by setting the OP code field. While rewriting the header fields, some values must be temporally be stored. In order to do so, the field *ig_md.addr_v4_buf* within the metadata is utilized. The case of the ARP responder shows how a table and action are implemented in P4. The remaining tables will only be explained without any more code or details. The next required table after the ARP table is the LPM table. The LPM table performs a longest prefix match on the destination IPv4 address and has two actions. One for a match where the packet will be sent out and the destination MAC address will be rewritten and one action for a cache miss which will send the packet to the forwarder.

```
1  // Answer arp for local bound v4 addresses
2  table t_l2_arp {
3
4          key = {
5              hdr.arp.dst_ip: exact;
6                  hdr.ethernet.dst_addr: exact;
7                  hdr.arp.opcode: exact;
8                  ig_intr_md.ingress_port : exact;
9          }
10         actions = {
11                 discard;
12                 arp_response;
13         }
14         default_action = discard();
15 }
16
17 action arp_response(mac_addr_t response_mac) {
18         ig_intr_tm_md.ucast_egress_port = ig_intr_md.ingress_port;
19         //Ethernet Manipulation
20         hdr.ethernet.dst_addr = hdr.ethernet.src_addr;
21         hdr.ethernet.src_addr = response_mac;
22
23         //Arp Response
24         hdr.arp.opcode = 0x0002;
25         hdr.arp.dst_mac = hdr.arp.src_mac;
26         hdr.arp.src_mac = response_mac;
27         ig_md.addr_v4_buf = hdr.arp.src_ip;
28         hdr.arp.src_ip = hdr.arp.dst_ip;
29         hdr.arp.dst_ip = ig_md.addr_v4_buf;
30 }
```

Listing 5.3: Table and action definition of the ARP responder from the cache's P4 program.

In order to rewrite the source MAC address of packets on egress, another table is required which will match on the packet's egress port. Then an action is provided to update the source MAC address. The last mandatory table will be the table that is required to forward the packets from the forwarder to the corresponding egress port. The table only has a default action which will remove the vlan tag and set the egress port id to the value that was stored in vlan header tag field.

To determine the control flow between the tables, the apply section of a P4 program is utilized. The apply section is presented in Listing 5.4. In line 2 it will be checked if an ARP header was parsed from the packet and if so, the ARP responder table will be applied. At line 5 the packet is checked for a vlan header. If a vlan header is present this indicates that the packet originates from the forwarder. The applied table in line 6 will remove the vlan tag and send the packet to the egress port having the same id as the vlan id. If the packet did not have a vlan header, it is checked for a valid IPv4 header in line 8. Afterwards a not yet introduced table will be applied in line 9. The table *t_l3_v4_exact* is introduced to perform an exact match on the destination address to forward packets belonging towards the router itself to the controller later. If the table does not match, the layer 3 LPM table is applied in line 10 which either sends the packets out directly or to the forwarder. Afterwards, the sampling of the ingress packets is performed. How the sampling works is explained later, but important is the location of the sampling within the pipeline. The sampling will only be performed on non-tagged packets. If all packets would be sampled, the tagged packets that are coming back from the forwarder would be sampled twice. This would lead to incorrect metrics. Finally, the last table *t_l2_src_mac* in line 20 is applied which will rewrite the source MAC address on egress.

```
1  apply {
2          if(hdr.arp.isValid()) {
3                  t_l2_arp.apply();
4          }
5          if (hdr.vlan.isValid()) {
6                  t_l2_res_sw_fwd.apply();
7          }
8          else if (hdr.ipv4.isValid()) {
9                  if(t_l3_v4_exact.apply().miss) {
10                         t_l3_v4_lpm.apply();
11                 }
12
13                 //sampling
14                 ig_md.sample = multi_counter_register_action.execute(0);
15
16                 if(ig_md.sample == 0x1) {
17                         send_to_mc_group(1);
18                 }
19         }
20         t_l2_src_mac.apply();
21 }
```

Listing 5.4: Apply section of the P4 cache program.

To perform the sampling two things are required: first a method to count the packets in order to identify the n-th packet and second a component to replicate the n-th packet to a mirror port. To count the packets, a register is used in P4. The register *multi_counter_register_action* can contain multiple counters. If the register is applied, it will execute the apply section of the corresponding code which is presented in Listing 5.5. The register will compare the current value with the sample rate in line 6. Once the counter has reached its target, it will reset the counter value to 0 in line 8 and return a 1 in line 7. Otherwise the counter is increased by one in line 11 and a 0 is returned. The counter is executed on every relevant packet in the apply section of the P4 program (Listing 5.4 line 14). If the execution returns a 1, the packet must be sampled, otherwise no sampling must be performed. To the replicate the packet, the multicast engine from the switch is used. In line 17 in Listing 5.4 the packet is sent to the group one of the multicast engine. The multicast engine of the switch is then configured to duplicate the

```
1  Register< bit<32>, bit<32>> (32w1) multi_counter_register;
2
3  RegisterAction<bit<32>, bit<32>, bit<1> >(multi_counter_register) multi_counter_register_action = {
4
5          void apply(inout bit<32> counter, out bit<1> read_value){
6                  if(counter >= (bit<32>)ig_md.sample_rate) {
7                          read_value = 1w0x1;
8                          counter = 32w0x0;
9                  } else {
10                         read_value = 1w0x0;
11                         counter = counter + 1;
12                 }
13         }
14 };
```

Listing 5.5: Counter implementation with registers in P4.

packet to the sampling port. This process of sampling using the multicast engine is inspired by P4STA [KSB+20], also the register code of P4STA was used and modified for our use case.

This section has described how the cache's P4 program was implemented. To interact with the P4 program from a controller an interface is required. The Tofino switch provides a generic gRPC [grp] interface which can be used to insert, update and delete entries from the tables. The use of gRPC to interact with P4 programs is the most common method today and the P4 language consortium also releases a specification for it [p4rb]. However, it must be mentioned that the gRPC API of the Tofino is proprietary so there is no guarantee that it meets the P4 language consortium's specifications.

5.1.2 P4HC Forwarder

After the implementation of the cache was provided, now the implementation of the forwarder will be presented. As mentioned in the design, the forwarder will be implemented using the DPDK framework. The DPDK version v20.05 was chosen at the beginning of this work, since it was the latest stable release at this time. To quickly implement the forwarder, it was chosen to use the l2fwd example [l2f] application as a starting point, which is provided with the framework. Based on the requirements, we stripped the example code down to a minimum code base in order to implement the forwarder. To sum up the requirements to the forwarder from the design: the forwarder must be able to perform a longest prefix match and be able to modify a vlan header. Also the LPM table must be able to store a full table.

The first step to move forward was to implement the ability to handle vlan tags within our DPDK application. To do so, the vlan offloading capability of the Intel XL710 NIC was utilized. This feature allows the parsing of the vlan header at the NIC level. Furthermore, the NIC will completely remove the vlan tag from the received packets and writes information like the vlan id to the metadata of the packet within the memory. In case of the DPDK framework, the vlan id is written to the corresponding *vlan_tci* field which belongs to the *mbuf* structure. The *mbuf* is the main data structure in DPDK which is used to describe stored packets in memory. This allows the application to access the vlan information without parsing the header itself. Once the packet was fully processed, the vlan tag can then be pushed again to the packet using the NIC's offloading capabilities on egress. The hardware offloading features are used to remove the effort of processing the vlan header in the application. This will safe additional CPU time and also implementation effort. Listing 5.6 shows which port configuration is required to enable vlan offloading at the NIC level within DPDK.

```
1  // global hardware configuration for offloading
2  static struct rte_eth_conf port_conf = {
3          .rxmode = {
4                  .offloads = DEV_RX_OFFLOAD_VLAN_STRIP
5          },
6          .txmode = {
7                  .offloads = DEV_TX_OFFLOAD_VLAN_INSERT
8          },
9  };
10
11 // has to be applied per packet to selectively enable offloading:
12 packet->ol_flags |= PKT_TX_VLAN_PKT;
```

Listing 5.6: Vlan offloading with the DPDK framework.

Line 2 shows the definition of the NIC configuration struct of DPDK which is used to configure the NIC on startup. Within this struct the offload functions can be defined for receive and transmit. Line 4 show the stripping of vlan tag on receiving and line 7 the addition of the vlan tag on sending the packet. The stripping of the vlan tag is performed automatically upon receiving a packet. The pushing of the vlan tag

on sending a packet will however not be performed automatically. In order to do so, the packet must be tagged with the *PKT_TX_VLAN_PKT* flag to instruct the NIC to push the tag back onto this packet (an example is provided in line 12).

After the vlan tag can be handled at the NIC level, the routing using LPM on a routing table must be implemented. The first attempt was to utilize the *rte_lpm* library provided by the framework itself. The library itself has worked fine, but we found some major disadvantages. Within the LPM table implementation it is possible to store an integer value in order to identify the next hop on a match. Our initial implementation based on the assumption to map the vlan id to the egress port was working as expected using the DPDK library. However, as we mentioned in the design, it is also required to store the new destination MAC address together with the port id in the LPM table to update the packet accordingly. This leads to the problem that it is not possible to store the MAC address and the vlan id within the DPDK LPM table itself. So a next hop table was implemented according to the design, which is also used within the controller. An array is used to store a custom next hop struct which is presented in Listing 5.7.

```
1  struct next_hop {
2      uint32_t nextHop;
3      uint32_t nextPort;
4      struct rte_ether_addr mac;
5  };
6
7  struct ctx_main {
8      ...
9      struct next_hop *ipv4_next_hops;
10     struct rte_fib* rte_fib;
11     ...
12 };
```

Listing 5.7: Main context data structure with the next hop array.

Line 1 presents the struct that is stored per next hop. It contains the next hop IPv4 address in line 2, the next port (port to send out at the cache; equals the vlan id) in line 3 and the destination MAC address in line 4. The array is then stored in the custom applications main context which is a struct called *ctx_main* in line 7. To quickly access the according next hop entry, the next hop's IPv4 address is used as the array index.

To identify the next hop on a match, the key of the next hop table is stored within the LPM table per prefix. At this point the *rte_lpm* library was not suitable for our use-case. The issue here was that the interface of the *rte_lpm* library exposes that an *unit32_t* can be stored as a next hop index within the table. While testing we found out that not all the 32 bits are useable, since the LPM data structure is using some bits for its internal housekeeping. This leads to the fact that in some use cases not all 32 bits of the stored next-hop id are returned upon a lookup. To solve this problem another LPM implementation of the framework was utilized which is called *rte_fib* . This library is still experimental at the time of writing, but has the advantage that it is possible to store a full *unit64_t* next hop index together with each prefix in the table. This now allows to store the full 32 bits of the next hops IPv4 address as an index for the next hop table. The data structure is finally stored within our main application context in line 9 of Listing 5.7.

To finally perform the LPM lookup on the packets and forward them, a function *route_v4* was implemented which is called on every packet. To understand the routing process, the function is presented in Listing 5.8.

At line 7 first the Ethernet header is parsed from the packet. Afterwards in line 9 it will be checked if the packet contains an IPv4 header and if so, the IPv4 header will be parsed in line 11. At line 18 the LPM lookup is performed. On a match, the LPM lookup will return the next hop index which is accessed in line 20. With the next hop key it is possible to lookup the correct next hop data for the packet within

```
1  static void route_v4(struct rte_mbuf *m, unsigned portid, unsigned queue)
2  {
3      struct rte_eth_dev_tx_buffer *buffer;
4      struct rte_ether_hdr *eth_hdr;
5      struct rte_ipv4_hdr *ipv4_hdr;
6
7      eth_hdr = rte_pktmbuf_mtod(m, struct rte_ether_hdr *);
8
9      if (RTE_ETH_IS_IPV4_HDR(m->packet_type)) {
10
11         ipv4_hdr = rte_pktmbuf_mtod_offset(m, struct rte_ipv4_hdr *, sizeof(struct rte_ether_hdr));
12
13         uint32_t dst_ip[1];
14         dst_ip[0] = rte_be_to_cpu_32(ipv4_hdr->dst_addr);
15
16         uint32_t nextHop;
17         uint64_t nextHopNew[1];
18         if (rte_fib_lookup_bulk(ctxMain.rte_fib, dst_ip, nextHopNew, 1) == 0) {
19
20             nextHop = (uint32_t) nextHopNew[0];
21             m->vlan_tci = ctxMain.ipv4_next_hops[nextHop].nextPort;
22             m->ol_flags |= PKT_TX_VLAN_PKT;
23             eth_hdr->d_addr = ctxMain.ipv4_next_hops[nextHop].mac;
24
25             // Send Packet
26             buffer = tx_buffer[portid];
27             rte_eth_tx_buffer(portid, queue, buffer, m);
28         } else {
29             // Drop packet
30             rte_pktmbuf_free(m);
31         }
32     }
33 }
```

Listing 5.8: Function route_v4 that does the lpm and rewrites the packet before forwarding.

the extra next hop table. In line 21 the according vlan id is set for the egress port of the cache and in line 22 the NIC is instructed to push the vlan tag on the packet. In line 23 then the destination MAC address of the packet is set to corresponding one from the new next hop. Finally, the packet is sent out in line 27. If the LPM lookup did not result in a match, the packet will be dropped at line 30.

After the data path of the DPDK application is implemented, now the control plane API is required to install the routes to the DPDK application externally. Since the design specifies that the API technology of the forwarder and the cache should be the same, gRPC [grp] is used. gRPC is also utilized at the cache with the Tofino switch, so it was the only choice. A custom gRPC API for the forwarder was implemented. In gRPC the interface is described in a language independent proto file, so that the API can be used with any programming language supported by gRPC. The proto definition of our API can be seen in Listing 5.9.

In the listing the structure of the most important parts can be seen. In line 1 the service and its methods are defined. As specified from the design, two methods will be required: one to add a new route and one to remove a route. As parameter each function gets the RouteIPv4 message which is described in line 6. The message itself contains the prefix and the corresponding subnet mask, the next hop vlan, the next hop IPv4 address and the new destination MAC address. For the method AddV4Route all parameters are required, while on the revocation only the parameters prefix and mask are required.

```
service RoutingApi {
  rpc AddV4Route(RouteIPV4) returns (Status) {}
  rpc RemoveV4Route(RouteIPV4) returns (Status) {}
}

message RouteIPV4 {
  uint32 prefix = 1;
  uint32 mask = 2;
  uint32 nextHopVLan = 3;
  uint32 nextHopIP = 4;
  bytes mac = 5;
}
```

Listing 5.9: gRPC service definition within the proto file.

Based on the proto file the code that is required to implement, the described service can be generated automatically for any supported language. However, the DPDK framework itself uses mostly C code for the forwarding code and at the moment there is no library available to generate gRPC services directly as C code. In order to use the gRPC service in our DPDK code, a wrapper must be implemented into a language that is supported by gRPC, which is C++. The trick to execute the C++ gRPC code from the C application is to define a wrapper class as extern C. This code can be called from the DPDK application. The required code can be seen in Listing 5.10.

```
//wrapper.h
extern "C" {

typedef struct GRPC GRPC;

GRPC* newGRPC();

void startServer(GRPC* grpc);

}

//Wrapper.cpp
extern "C" {

    GRPC* newGRPC() {
        return new GRPC();
    }

    void startServer(GRPC* grpc) {
        grpc->startServer();
    }

}
```

Listing 5.10: Wrapper code that is used to start the C++ gRPC API in a C program.

The extern statement allows a C application to access the specified C++ code. Since C++ is object oriented and C is not, the wrapper must be able to translate between the two languages. The wrapper will provide the C++ class as a struct to C. To access the methods of the gRPC class, wrapper methods must be implemented since C cannot directly access a classes' member methods.

Within this section the most important parts of the forwarder implementation were provided. In the next section the implementation of the metric collector is shown and afterwards the controller will be described.

5.2 Metric Collector

Within the design it was specified that generating metrics for the offloading decision will be done by analyzing packet samples from the cache. It was worked out which task and steps a collector has to perform in order to provide metrics per prefix in bits per second and packets per seconds. Afterwards, the metrics need to be stored within a time series database, so the controller can access the metrics.

Since the implementation of such a collector would be a major task itself, an open-source collector called pmacct [pma] was used. Pmacct is an universal collector for network monitoring. It can evaluate many kinds of flow samples like IPFIX [ZCQZ04], sFlow [PMP01] or - even more important for us - an option to collect packets through libpcap [lib] over a physical NIC port. Pmacct itself has multiple daemons for each kind of input data. In case of the libpcap input stream, the pmacctd tool must be used. The tool itself can then be configured to listen on a given interface to collect incoming packets. Also pmacct has all required functions for the aggregation of the data built-in. It supports the aggregation of data on most fields of a packet header and can even include external meta data. In our use case it is required to aggregate the packets based on the destination IPv4 address from the packet header, combined with the matching prefix in the routing table. In order to generate this kind of data, the pmacctd daemon must be provided with all available prefixes within the network. A simple way would be to provide pmacctd a file with all available networks. However, this approach would not be sufficient since the prefixes can change at any given time. This would require the controller to write a new file with all prefixes on every update and forces pmacctd to reload afterwards.

To solve this issue the pmacct toolsuite also comes with a BGP daemon. The BGP daemon allows to send the full routing table down to the pmacctd collector. Pmacctd then has the ability to aggregate the data based on the prefixes that were provided by BGP.

As the data is aggregated internally within the collector, the data must be normalized as noticed in the design. The received packets are only sampled at a given rate. Pmacctd already has the ability to renormalize the sampled data by providing the sample rate in its configuration file.

Until now pmacctd can provide any functionality that was required from the collector within one tool. The last remaining step is to read out the data from pmacctd and store it to a time series database. The easiest way to read the data from pmacctd is to use the print plugin. The print plugin writes a JSON [Bra17] file in a user specific interval with the aggregated data over the selected timeframe. After the data is written, pmacctd can call an external script which will process the data afterwards. A script is used to read the data from the file and push it into a time series database. An example output of the pmacctd collector is presented in Listing 5.11.

```
{"event_type": "purge", "net_dst": "10.0.10.3", "mask_dst": 32, "packets": 3294490, "bytes": 48622470}
{"event_type": "purge", "net_dst": "10.0.10.2", "mask_dst": 32, "packets": 9810300, "bytes": 95423000}
```

Listing 5.11: Example JSON output of the pmacctd daemon.

In the example output each line represents the aggregated metrics per prefix. The prefix is represented by the net_dst and mask_dst fields. Afterwards the metrics in form of packets per second and bytes per second are provided. In the implementation of P4HC a metric file like this is generated every 10 seconds. A 10 second interval was chosen to make sure that pmacctd has seen enough packets to generate the right metrics. If smaller windows are selected, pmacctd processes fewer packets which can lead to incorrect metrics on smaller flows.

As a time series database InfluxDB was selected. InfluxDB [infa] is open source and easy to use. Also InfluxDB has the advantage that data can be queried using a SQL like language called InfluxQL [infd]. InfluxQL allows to query the data based on any attribute and also selection- and aggregation-operations like MAXIMUM or TOP N are provided. This allows us later on to determine the prefixes with the highest volume based on an InfluxQL query. So the effort can be reduced to writing a simple query that selects the most active prefixes from the raw data and performs the required selection and aggregation on the data. In InfluxDB the data can be stored in tables. Within a table, a row can be written. Each row has a timestamp, some data fields and can have additonal tags. Tags are used to map data fields to certain attributes. In our case we use the tag to mark a row with the corresponding prefix. An example is provided in listing 5.12.

```
name: prefixes
--------------
time                 bytes          pps            prefix
2020-10-29T19:13:41Z 50800364600    34580900       10.0.10.2/32
2020-10-29T19:13:41Z 12808260200    12291100       10.0.10.3/32
2020-10-29T19:13:51Z 30568724100    20712700       10.0.10.2/32
2020-10-29T19:13:51Z 5880645100     5996900        10.0.10.3/32
```

Listing 5.12: Example table structure used by InfluxDB.

In the example, the table which stores the data from pmacctd can be seen. In the first column the timestamp for each row can be seen. Afterwards the bytes per second and packets per second are shown. To tag each data set to a given prefix, the last column is used to write the prefix as a custom tag to each row. In the example can be seen that there is a data point for each prefix every 10 seconds. In order to get the data every 10 seconds into InfluxBD, a small python [pyt] script was written that parses the JSON file from pmacctd and pushes the data to InfluxDB by utilizing the InfluxDBClient [infb] library, which is provided by InfluxDB itself.

To summarize the collector, Figure 5.2 provides an overview on how the sampled packets are evaluated and stored into InfluxDB.

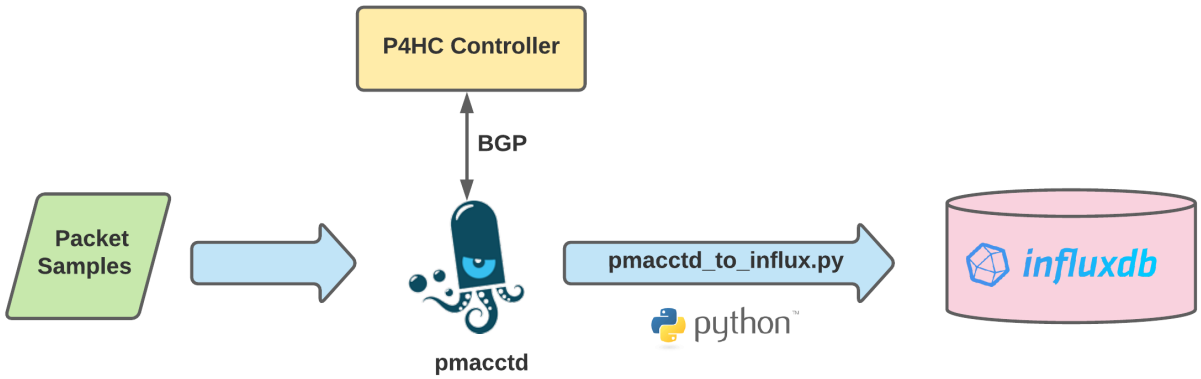


Figure 5.2: Metric collector setup.

5.3 P4HC Control Plane

The last part of the P4HC puzzle that must be implemented is the controller itself. To start the implementation, a suitable programming language had to be selected. It was decided to implement the P4HC

controller using the Go language [go]. Go was chosen because all relevant libraries that were required are available for Go and it offers a solid gRPC support for interacting with the data plane. Also Go enables simple usage of concurrency and offers decent performance due to its C-like approach. Finally, Go compiles the complete program into one binary which can then be executed on a server easily without installing any dependency libraries.

Since the controller consists of multiple components, the section is also divided into multiple logical subsections. First the implementation of the north and southbound interface is described. Second the RIB itself will be presented. Afterwards the most important configuration parts are introduced and the start up sequence of the controller will be explained. Finally the offloader is presented.

5.3.1 North- and southbound interfaces

The two major interfaces described within the design are the northbound interface, which should be BGP, and the southbound interface to the data plane in gRPC.

The northbound interface itself must be able to receive the routes and add them to the RIB. As mentioned before, the interface must speak BGP in order to exchange the routes with the neighbor routers. As Go is used for the controller a BGP implementation in Go is required. The bio-rd [bio] library was chosen. Bio-rd is a library which implements multiple routing protocols (BGP, IS-IS and OSFP) and also has implemented its own RIB. This allows us to implement BGP, but also other protocols could be supported with minimum effort. We implemented a BGPServer class that contains the bio-rd library, configures the neighbor peers and starts the BGP Server in a second thread. While configuring the BGP neighbors from the configuration, it is also required to setup the pmacctd collector as a neighbor. Since bio-rd comes with its own RIB and works like a full-featured router without a data plane, all available routes are sent down to the collector, if its autonomous system number differs from the controller one. Also bio-rd supports input and output policies for the BGP neighbors, so it possible to filter routes on import and export. To mirror the routes from the bio-rd RIB to our own RIB, we have implemented an interface from the bio-rd library called RoutingApiClient. This interface contains methods that are called from bio-rd on route updates. We implemented this interface in a new class which also contains a reference to our RIB. So at each update within the bio-rd RIB we can synchronize the routes with our RIB.

```
1  type MyRouteTableClient struct {
2          client protos.RoutingApiClient
3          rib    *Rib
4  }
5
6  func (myRouteTableClient MyRouteTableClient) AddPath(pfx *bnet.Prefix, path *route.Path) error {
7          myRouteTableClient.rib.AddRoute(*pfx, path.NextHop().ToNetIP())
8          return nil
9  }
10
11 func (myRouteTableClient MyRouteTableClient) RemovePath(prefix *bnet.Prefix, path *route.Path) bool {
12         myRouteTableClient.rib.RemoveRoute(*prefix, path.NextHop().ToNetIP())
13         return true
14 }
```

Listing 5.13: Custom RoutingApiClient to extract the routes from the bio-rd RIB and send them to the controller's RIB.

The most important parts of the class can be seen in Listing 5.13. Line 1 shows our custom type which contains a reference to the controller's RIB and contains the interface of the bio-rd RIB. At startup the type is instantiated and passed to the bio-rd RIB. In line 6 the addition of a new route is shown and in

line 11 the revocation of a route. This design allows to replace bio-rd in future, if needed, by just adding a new BGP library and pass the routes into the controller's RIB.

The southbound interface consists out of two interfaces. One to the cache and one to the forwarder. As the design describes, it should be possible to exchange both data planes later. At the moment both data planes utilize gRPC as API. However, since even in gRPC the service definitions can differ, we decided to add two generic interfaces in Go: one for the cache and one for the data plane. This allows to plug in any data plane to the controller with any API, even non-gRPC, if desired. The interfaces are called ICache and IForwarder and are designed according to the API interfaces specified by the design. Listing 5.14 presents both interfaces.

```
1  type IForwarder interface {
2
3          AddRoute(prefix bnet.Prefix, nextHopPort int, nextHopMac net.HardwareAddr, nextHop net.IP) bool
4          RemoveRoute(prefix bnet.Prefix) bool
5  }
6  type ICache interface {
7
8          AddRoute(prefix bnet.Prefix, nextHopPort uint32, nextHopMac net.HardwareAddr) bool
9          RemoveRoute(prefix bnet.Prefix) bool
10
11         AddIPAddrToPort(ip net.IP, port uint, mac net.HardwareAddr) bool
12         DelIPAddrFromPort(ip net.IP, port uint) bool
13
14         SetSrcMacToPort(mac net.HardwareAddr, port uint) bool
15         DelSrcMacFromPort(port uint) bool
16
17         SetForwarderPort(port uint) bool
18         SetSamplingPort(port uint32) bool
19 }
```

Listing 5.14: Data plane interfaces of the P4HC controller.

Both interfaces are implemented in new classes called dpdkForwarder and tofinoCache. The implementation of the forwarder interface was straight forward, since it was essentially a wrapper for the gRPC interface of the forwarder. The implementation of the interface for the Tofino was more effort, since at the first connection a start-up sequence must be run on the switch through gRPC. Afterwards the update messages for the hardware tables must be constructed. The gRPC interface of the Tofino is generic and provides its own protocol on top of gRPC. As the Tofino-Interface is the more complex case, it will be described in more detail in the following.

The first step is to connect via gRPC to the Tofino device. Afterwards, a gRPC streamChannel must be opened over which then the messages for the initialization can be sent. Three steps are required to perform the initialization. At first a Subscribe message must be sent to the switch with the id of the switching chip. In our case the switch contains only a single chip, so we can pass 0 as id. Afterwards a SetForwardingPiplineConfigRequest must be performed. Within the request, the name of the P4 program that the client wants to communicate with is passed. Afterwards the final step is to request the ForwardingPiplineConfig. The ForwardingPiplineConfig consists of two JSON files which represent the switch's tables in a generic way. One JSON contains all tables from the executed P4 program and one contains the tables that are provided by the switch like the ones for the multicast engine, for example. The information from these JSON objects is required in the next steps in order to create the messages to update the entries of the tables.

Listing 5.15 shows an excerpt of the table definition from the P4 runtime JSON. All non-necessary attributes have been removed in order to simplify the example. Each table is represented by a JSON object. In the example the table t_l2_src_mac was selected, which updates the source MAC address of

```
1  {
2    "tables": [
3      {
4        "name": "pipe.SwitchIngress.t_l2_src_mac",
5        "id": 33577985,
6        "key": [
7          {
8            "id": 1,
9            "name": "ig_intr_tm_md.ucast_egress_port",
10           "match_type": "Exact",
11           "type": {
12             "type": "bytes",
13             "width": 9
14           }
15         }
16       ],
17       "action_specs": [
18         {
19           "id": 16824300,
20           "name": "SwitchIngress.set_src_mac",
21           "data": [
22             {
23               "id": 1,
24               "name": "src_mac",
25               "type": {
26                 "type": "bytes",
27                 "width": 48
28               }
29             }
30           ]
31         }
32       ]
33     }
34   ]
35 }
```

Listing 5.15: Runtime JSON from the Tofino which describes the table formats for the interaction with the switch.

a packet. The table has an unique id in line 5 which is required for an update message later on. Also in the key section (line 6), all keys including their matching behaviors are listed. In this example there is only one key which performs an exact match to the egress port. If a table matches on multiple fields, the key array will consist of multiple key objects. When inserting a new entry into the table, an action must also be specified. The available actions for a table are described in the action_spec section (line 17). The action has an unique id and data fields. The data field describes the parameters of this action. In this case there is only one parameter for the action, which is the source MAC address. However, it is important to understand that at this point the definition does not enforce the passed argument to be an actual MAC address. At lines 25-26 can be seen that the switch only requires the type to be bytes and that is expecting exactly 48 bits of data. This unawareness of the representation allows the format to fit any P4 program. However, this also makes the interaction with the device more complicated.

Based on the information from the JSON defintions, a message can be constructed that can be sent via the gRPC API to the switch in order to create a new table entry. The key parts are presented in Listing 5.16. The example shows that it is now required to rebuild the JSON structure of the table using the gRPC message formats. At line 1 the specified key must be rebuild, important is that the key id matches the FieldId. Afterwards the correct matcher type must be specified. The JSON from the example before

```
var fields = []*bfruntime.KeyField{
  &bfruntime.KeyField{
    FieldId: 1,
    MatchType: &bfruntime.KeyField_Exact_{
      Exact: &bfruntime.KeyField_Exact{
        Value: out_port,
      },
    },
  },
}

var data = &bfruntime.TableData{
  ActionId: 16824300,
  Fields: []*bfruntime.DataField{
    &bfruntime.DataField{
      FieldId: 1,
      Value: &bfruntime.DataField_Stream{
        Stream: dst_mac,
      },
    },
  },
}

var myEntry = bfruntime.TableEntry{
  TableId: 33577985,
  Key: &bfruntime.TableKey{
    Fields: fields,
  },
  Data: data,
  IsDefaultEntry: Flase,
}
```

Listing 5.16: Definition of a table entry to create a gRPC update message.

shows that the table requires an exact match, so the corresponding message for an exact match must be passed. Within the exact field, the value that will be matched must be provided. In this case the value is the egress port id, formatted as a byte array. After the key is defined, the action that will be performed on an match must be defined. The action is defined in the data section in line 12. First the action id from the JSON must be specified and afterwards the parameters (lines 14-21). Finally the key and the action must be matched to the table. Line 24 shows the definition of the table entry. The definition requires the table id from the JSON in line 25. Afterwards the predefined key and action data must be inserted. The created message can then be sent with an update message to the switch using a gRPC write request. This case has shown the basics on how an interaction with a Tofino switch looks like. In order to implement the interface, the update messages must be defined for each method.

This section has provided in-depth details on how the data planes' APIs are implemented. In the next section the implementation of the RIB will be presented.

5.3.2 Routing Information Base

In this section the implementation of the RIB is presented. The RIB itself will contain all available prefixes and the next hop table. Also, the RIB coordinates the interaction with the forwarder and the cache.

The main data structures of the RIB can be seen in Listing 5.17. To store the next hop table, a map was implemented, which uses the next hop IPv4 address as an index (line 2). The map uses the custom

```
1  type Rib struct {
2    nhTable   map[uint32]nextHop
3    tree      *nradix.Tree
4    fwd       forwarder.IForwarder
5    cache     cache2.ICache
6    offloads   map[string]bnet.Prefix
7  }
8
9  type nextHop struct {
10   IP   net.IP
11   Port uint32
12   Mac  net.HardwareAddr
13 }
14
15 type ribEntry struct {
16   NextHop   uint32
17   IsInFwd   bool
18   IsInCache bool
19   Prefix bnet.Prefix
20 }
```

Listing 5.17: Data structures of the RIB implementation.

struct nextHop (line 9) to store the corresponding egress port and destination MAC address. The table will be filled automatically on startup, based on the configuration.

The prefixes itself are stored, as specified in the design, within a radix tree. To realize such a radix tree, the nradix tree [nra] library was used, since it was the only suitable library available. However, since even this library missed some functionality that was required, we copied the source file in order to add new methods to the library. We first tried to extend the library, but there is no easy way in Go to extend a struct with access to the internal variables of the library which are necessary. The tree itself allows to store any struct besides a prefix. To store information such as the next hop, we created a new struct ribEntry (line 15) that stores a reference to the next hop table, the information whether the prefix is located within the cache or forwarder and the prefix itself. In certain situations it was not possible to extract the prefix after a lookup within the tree library. Finally, the RIB contains an instance of the cache (line 5) and forwarder (line 4) API, as well as a list off all prefixes which are installed within the cache (line 6). This list contains information which can be extracted from the tree, but the extraction would be too much effort while creating the delta map later on, so it was decided to store this information here too.

If a new route is added to the RIB, the RIB will lookup the next hop within the next hop table and afterwards installs the route into the forwarder and the radix tree. While inserting the prefix into the tree, the prefix will be flagged as installed into the cache. On deletion of a route, it will at first be checked if the route is within the cache, if so is removed, and afterward is deleted from the forwarder and the radix tree. To install a prefix into the cache, a method is provided to do so and one to remove. It must be noticed that this method only moves this specific prefix into the cache and does not perform any dependency checks.

In order to offload a set of prefixes into the cache, a OffloadBulk method is provided, which will also resolve the dependencies and make sure that the cache will not be overloaded. Since the method protects the cache from overloading, the provided prefix array should be sorted according to their priority, because on resolving of the dependencies some entries may not be installed into the cache. The code excerpt to resolve the dependencies and protect the cache can be seen in Listing 5.18.

The code implements the behavior of the design. At first in line 1, an iteration over the prefixes which are offload candidates is performed. For each prefix all dependencies will be resolved. The

```
1  for k, prefix := range candidates {
2
3    // Resolve dependencies
4    dep, err := r.tree.FindAllMoreSpecificCIDR(prefix.String())
5
6    if err == nil {
7
8      if (currSize + len(dep) + 1) > cacheSize {
9        continue // Cache is full, continue in order to find a smaller set, if possible
10     }
11
12     for j, v := range dep {
13       e := v.(ribEntry)
14       dependencies = append(dependencies, e.Prefix)
15       _ = j
16     }
17
18     prefixes = append(prefixes, prefix)
19     currSize = currSize + len(dep) + 1
20   } else {
21     continue
22   }
23   _ = k
24 }
```

Listing 5.18: Resolving of dependencies and cache size alignment.

method FindAllMoreSpecificCIDR will search for the prefix within the radix tree and return all successor-nodes which are prefixes, as described in the design. As this method was not available, it was custom-implemented by us. After the dependencies are resolved in line 8, a check will be performed if the prefix including its dependencies will fit into the cache. The value *currSize* represents the number of entries that are to be installed into the cache by previous iterations. Then the number of dependent prefixes is added and the prefix itself is added. If the sum of entries is less than the actual cache size, the prefix and the dependencies can be further processed. If they do not fit the cache, the next iteration can be forced, since there is no space left for the current prefix. If another, smaller prefix fits it will be added to the list of prefixes. Finally, the new currSize is calculated (line 19).

After the resolving the dependencies, the list of prefixes and dependencies will be compared to the currently installed prefixes. As the design describes, the result of the comparison is a list of prefixes that must be installed into the cache and one which must be removed. The two list are sorted by prefix length before being applied to the cache. At first the prefixes will be removed in order to prevent overfilling the cache or incorrect forwarding behavior. Afterwards the new prefixes will be applied to the cache.

This section described the main parts of the RIB implementation. In the next section the configuration will be provided.

5.3.3 Configuration

The configuration of P4HC is realized by a JSON file which contains all necessary information for the controller. Within the configuration file, there is connection information to the cache and forwarder stored, the location of pmacct and so on. The most important and noticeable parts are the port configuration for the cache, the next-hops and the BGP neighbors. Those three parts are presented in Listing 5.19.

```
{
  "ports": [
    {
      "port_id": 389,
      "mac": "03:08:09:01:02:03",
      "ips": [
        "10.0.0.1"
      ]
    }
  ],
  "next_hops": [
    {
      "ip": "192.168.1.1",
      "dst_port": 389,
      "dst_mac": "3c:fd:fe:9e:14:f9"
    }
  ],
  "bgp": {
    "neighbours": [
      {
        "peer_as": 65001,
        "peer_ip": "1.1.1.1",
        "import": true,
        "export": false
      }
    ]
  }
}
```

Listing 5.19: Subsection of the configuration file.

The first part is the port configuration. The list of ports contains the information which port at the cache is used. To identify the port, the switch's internal id is used. Also the MAC address for the port is specified. The MAC address will be used to rewrite the source MAC address once a packet leaves this port and is also used within an ARP response. The IP addresses are also specified within the port configuration.

In the next part all available next hops can be defined. A next hop contains the egress port and a destination MAC address. The IP address of the next hop is then used to map the next-hop to a BGP route and as internal identifier. On startup, the next hops are read and stored in the next-hop table within the RIB.

The last part is the BGP neighbors configuration. Within this section all available BGP peers will be defined. In order to establish a BGP session, the peer autonomous system number and IP address must be specified. Also, it can be adjusted whether routes should be accepted and if the peer should receive routes.

5.3.4 Startup

At the startup of the controller all components must be initialized, as well as the data planes. If the controller first starts, the logger is initialized. For logging the Logrus [log] library is used. After logging is set-up, the next step is to read the configuration file. Afterwards the connection to the cache is established and the described initialization sequence on the Tofino is performed. When the connection is ready, the cache data plane is configured based on the given configuration file. At first the ports are initialized, which means that the source update on egress will be configured and the IP addresses for

the ARP resolver are programmed. As the last step the multicast groups for sampling are configured. After the cache's data plane is initialized, the connection to the forwarder will be established. At the forwarder, no extra initialization sequence must be performed. Now the RIB is instantiated. The RIB gets the information on how to reach cache and forwarder and builds the next hop table, based on the configuration file. Once the RIB is ready, the BGP daemon will be started. At first the local IP and AS is set up and the BGP session to the pmacctd collector will be established. Then all configured BGP neighbors are added and, if static routes were configured in the config file, they are loaded aswell. Finally, the BGP daemon is started within a new thread. At the last step, the offloader and the timer are initialized. The offloader itself will be described in the next sections.

Important is that before the controller itself is started, all external components should be ready. This includes the forwarder's DPDK application, the Tofino switch, the pmacctd daemon and the InfluxDB, as well as the BGP neighbors. Another important pre-requirement is that all data planes have to be fully resetted before the controller startup, since previous configurations could interfere with the startup sequence.

5.3.5 Offloading

The last piece of the controller will be the offloader itself. The offloader itself must identify the prefixes which should be installed to the cache. If the prefixes are selected, they must be handed over to the RIB. The RIB will then handle the cache management, as well as the dependency management. To identify all prefixes, an InfluxDB client class called influxClient was used. To connect and query the InfluxDB, the official Go library was used [infc]. In order to find the most active flows from the last minute, we have written a query which selects the top *n* active prefixes per time bin. The query can be seen in Listing 5.20.

```
q := client.NewQuery(fmt.Sprintf("SELECT top(\"bytes\", %d) FROM \"prefixes\"
WHERE time >= now() - 1m GROUP BY time(10s), \"prefix\" fill(null)", flux.topN), flux.db, "ms")
```

Listing 5.20: Influx query to get the n most active prefixes for the last minute.

The query selects the *n* most active prefixes during the last minute. As the top function returns the top *n* prefixes per time bin - in this case 10 seconds - we can get up to six different results per prefix in one minute. To normalize this data all results are aggregated and a median is computed. The offloader then sorts the list on the median metric, so that the most active prefixes are located on top. The list is then passed to the bulkoffload method from the RIB, which will compute all dependencies and installs them into the cache. The value for *n* is selected accordingly to the specification from the design, representing the cache size.

As the offloader must run continuously, it is started in a separate thread and is invoked every 10 seconds to generate a new set of prefixes.

5.4 Summary

In this chapter the implementation of the P4HC prototype was provided. To summarize, we have implemented a cache data plane in P4 which runs on a Tofino switch, a forwarder data plane in DPDK which runs on a commodity x86 server, a metric collector based on open source available tools and a custom controller written in Go. While implementing the P4HC we have also focused on modularity, so many components can be exchanged or upgraded in the future. The complete design was finally set up in a test environment, which is described in Figure 5.3.

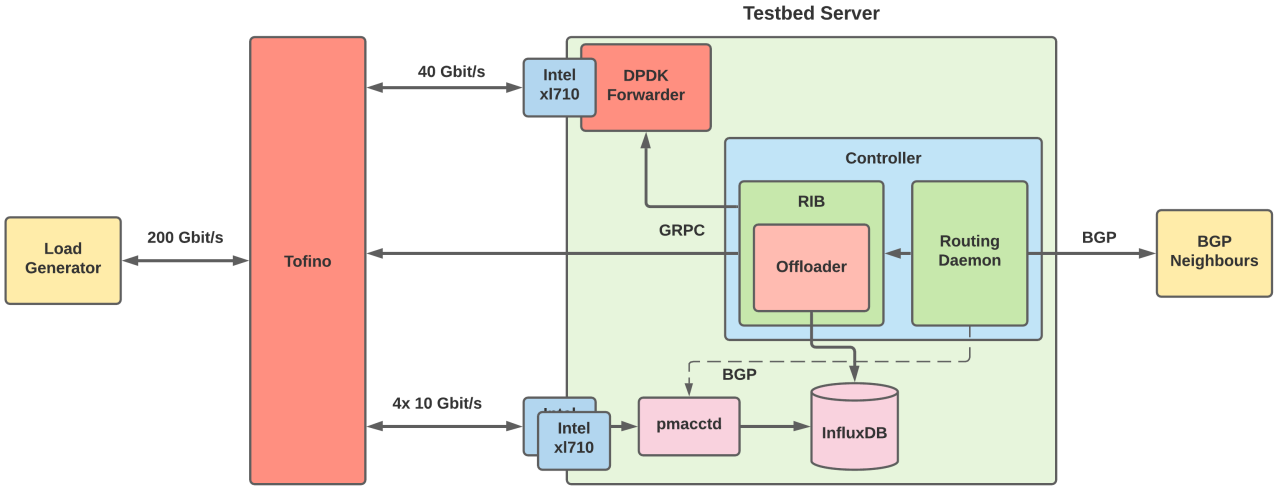


Figure 5.3: Prototype setup for the testbed.

As hardware switch we utilized a WEDGE 100BF-65X based on the Tofino chip. A x86 server with dual Intel Xeon E5 2650v4 CPUs, 256 Gb of system memory and two 40 Gbit Intel XL710 NICs. The forwarder is connected with 40 Gbit/s to the Tofino switch. The second NIC is split into four times 10 Gbit/s ports, where one port is used to receive the samples. To then test and benchmark P4HC we connected a load generator, which is capable of saturating a 100 Gbit/s link. How well P4HC performed can then be evaluated in the next chapter.

6 Evaluation

In the last chapters, the full design of P4HC was introduced and the implementation of the prototype was described. In this chapter the prototype will be evaluated. The focus of the evaluation will be on the data plane, as this was the key point of this work. More precisely, the performance of the data plane is evaluated in detail. As the data plane of P4HC consists out of two implementations for different underlaying hardware technologies, the cache and the forwarder, there are three cases that must be evaluated. These three cases describe how packets are traveling to the data plane in P4HC. In the first case a packet is only forwarded by the cache. This means that a layer 3 route is installed into the cache to forward the packets directly on the hardware switch. The second case is when there is no route installed into the cache and the packets must be routed by the forwarder. The packets must traverse the cache first and are then processed by the forwarder. At the forwarder, the routing is performed and the packet is sent back to the cache, which will send the packet out through the corresponding port. When we evaluate this case, we will essentially evaluate the forwarder, since it will become the bottleneck in the case. However, it must be noticed that the cache is still involved in this scenario. The last case is the hybrid one: the hybrid case describes the situation where a route is installed to the cache to offload traffic from the switch. In this situation there is a transition window that must be evaluated, since the movement of the packet flow could lead to side effects, as we will present later in this evaluation. In all three cases the performance of the data planes are the key point to investigate. To classify and compare the performance, this evaluation will focus on the three performance metrics: How many packets per second (pps) can be forwarded, how many bits per second can be processed (bit/s) and the latency that is introduced to a packet when traversing the data plane.

However, there is one additional part that needs to be evaluated. These three cases are focusing essentially the raw forwarding performance. However, with P4HC it is also important how fast and how often the data plane tables can be adjusted. Since one of the P4HC key tasks is to move routes from the forwarder to the cache on demand, in order to protect the forwarder from overloading. As the update rate of the data plane tables also require controller interaction, we will evaluate the full control path down to the data plane as well. This means we will evaluate the performance from the controller's point of view as the control channel of the data planes.

The evaluation will be structured as following: first the testing setup is presented that is used to evaluate the data planes' performance. Second all three cases for the data plane are evaluated. Afterwards the design for the control path evaluation is presented and the results are examined. Finally the results will be summed up.

6.1 Evaluation Setup

The previously mentioned testing scenarios will now be evaluated. To perform this evaluation, P4HC must be set up within a test environment and must be connected to a load generator. The setup is depicted in Figure 6.1. On the left side, the load generator is presented and on the right side the P4HC core components under test. As load generator P4STA [KSB+20] is used. P4STA is a load generator capable of generating large traffic volumes, while measuring the exact latency of each packet. To do so, P4STA also utilizes an additional P4 programmable Tofino switch. This Tofino switch enables P4STA to timestamp each packet that traverses to the Device Under Test (DUT), which is P4HC in our case. With the help of the P4STA Tofino, the packets that are sent to the DUT are timestamped within the egress pipeline of the P4STA Tofino. After traversing P4HC, they will be stamped again at the ingress pipeline by the P4STA Tofino. This allows exact latency measurements, since the packets are timestamped at the last possible point of the load generator. To further analyze the timestamps, it is possible to send the received packets to an external host. Here both timestamps can be read from the packets and the latency can be

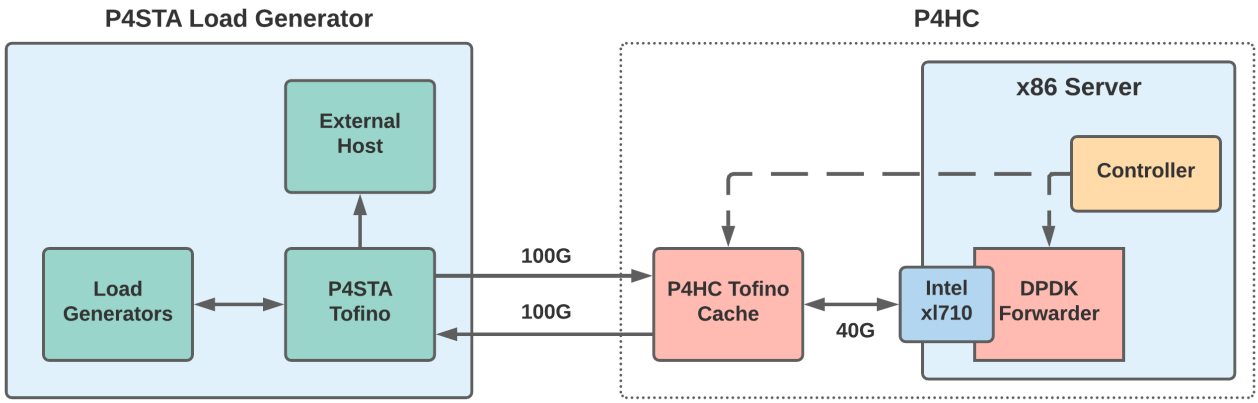


Figure 6.1: Test setup of the P4HC evaluation with P4STA as load generator.

computed. The timestamp precision is in the range of a few nanoseconds, which is suitable for testing P4HC, since modern 100G switches have a forwarding latency of around 450 nanoseconds [x70]. To generate the load packets, which are timestamped by the P4STA Tofino, Iperf3 is used. Iperf3 allows to generate one or multiple TCP or UDP flows between a client and server. Since the performance of IPerf3 would not be sufficient to saturate a 100 Gbit/s connection, the P4STA Tofino is also used to duplicate the Iperf3 packets in order to increase the load to the DUT. When the duplicated packets are received again, the P4STA Tofino will not send them back to the Iperf3 server, so that Iperf3 won't notice that the packets were duplicated and thus not interfere with the legitimate Iperf3 flows. To rate limit the load to the DUT, P4STA is also capable of hardware rate limiting the load using the P4STA Tofino's based shapers. Finally, as the load increases, more timestamp packets are generated and sent to the external host. Since high packet counts can overload the external host, it is possible to down-sample the packets that are sent to the external host on the P4STA Tofino. However, if every packet should be sampled, P4STA also provides an external host written in DPDK. This in turn generates large data sets and should only be used if needed.

To test P4HC, P4STA is connected with two 100 Gbit/s links to P4HC. Over the first link the packets are sent to P4HC and over the second link the packets are received again. The 100 Gbit/s links are connected to the cache device of P4HC, which is in this case also a Tofino switch (WEDGE 100BF-65X [wed]). From the P4HC Tofino a 40 Gbit/s link is connected to the server, which runs the DPDK forwarder. In addition, a second 10 Gbit/s link is connected to send the sampled packets from the P4HC Tofino to the metric collector. This link is not explicitly shown in the figure. The server is dual socket system with two Intel Xeon E5 2650v4 [i26] processors and 256 GB (DDR4 2400 MHz) of system memory. As NIC for the forwarder a 40 Gbit Intel XL710-QDA1 [xl7] was used. To connect the 10 Gbit/s link for the packet samples, also an Intel XL710-QDA1 NIC was used operating in breakout mode and thus allowing to split up the 40 Gbit/s port to four times 10 Gbit/s. Finally, also the controller of P4HC, including pmacctd and InfluxDB, was executed on the same server as the forwarder.

This section describes the testing setup to evaluate P4HC, however since it is required to test different cases that can each require different loads, the settings of the load generator will be presented in each section individually.

6.2 P4HC Cache Evaluation

After the test setup was described, the first case that needs to be evaluated is the P4HC cache's performance. However, before the performance of the cache is measured, a short branch is made to the Tofino's hardware internals to solve the question of how many routes the cache can handle.

6.2.1 Tofino Hardware Utilization

In the last chapters it was introduced how a Barefoot Tofino switch is used to execute the P4 program of the cache. Now the question is how many routes the cache can handle. To answer this question, it is necessary to understand how the Tofino switch maps a P4 program to its internal resources. When a P4 program is compiled to the Tofino switch, the tables of the program are mapped to the internal stages of the switch. The switch has internally 12 stages with a fixed number of resources available. The compiler places each table from the P4 program to a stage. To optimize the available space, the compiler can place multiple tables into one stage or stretch a table over multiple stages, for tables that cannot fit into one stage. When the compiler places the tables to the stages, it is generating a report file that provides the information which table is realized in which stage and how many of each resources are utilized per stage. While investigating the compilers results, we could observe that the compiler has realized the LPM cache table in the TCAM of the switch. TCAM is Ternary Content Addressable Memory, which allows to perform wildcard matches on a packet at one operation, on all its entries [ACS03]. So the limiting resource is the available TCAM space of the Tofino switch in this case. At the fist try, there were many stages not being used within the Tofino. We increased the size of the layer 3 LPM table step by step, while observing the placement within the stages. As expected, the compiler begins to stretch the LPM table over multiple stages. While increasing the table size, we noticed that it was possible to stretch the LPM table over 9 of the 12 stages. We have reached the limit at 110.592 entries. By then the TCAM utilization of all 9 stages was at 100 percent. The output of the hardware allocation can be seen in Listing 6.1. It must be noticed that we have omitted all other hardware allocation metrics, because the output would otherwise be too long.

```
-------------------------------------------
| Stage Number |  SRAM | Map RAM |   TCAM  |
-------------------------------------------
|      0       | 6.25% |  0.00%  |  0.00%  |
|      1       | 7.50% |  4.17%  |  0.00%  |
|      2       | 8.75% |  0.00%  | 100.00% |
|      3       | 8.75% |  0.00%  | 100.00% |
|      4       | 8.75% |  0.00%  | 100.00% |
|      5       | 8.75% |  0.00%  | 100.00% |
|      6       | 8.75% |  0.00%  | 100.00% |
|      7       | 8.75% |  0.00%  | 100.00% |
|      8       | 8.75% |  0.00%  | 100.00% |
|      9       | 8.75% |  0.00%  | 100.00% |
|      10      | 8.75% |  0.00%  | 100.00% |
|      11      | 2.50% |  0.00%  |  0.00%  |
|              |       |         |         |
|   Average    | 7.92% |  0.35%  |  75.00% |
-------------------------------------------
```

Listing 6.1: Hardware resource allocation of the P4 cache program on the P4HC Tofino switch for 110.592 IPv4 routes.

6.2.2 P4HC Cache Throughput

After it was investigated how many routes the cache can handle, the performance of the cache must be measured. As we introduced before, the following three metrics will be used: packets per second, bits per second and the latency. However, since those are only the three metrics, it must of course be defined under which conditions the measurements are taken. The most influencing factor to the

performance of packet forwarding is the size of the forwarded packets itself. So, it was decided to measure the performance of the data plane at different packet sizes. A test was designed which measures the performance of the data plane at 15 different packet sizes, beginning from 100 bytes per packet, up to 1500 bytes, in 100-byte steps. We selected this interval due to the following reasons: first 1500 bytes usually represents the maximum size that packets can be within the Internet, since Jumbo Frames are often not supported by carriers. Second, the lower limit to 100 bytes was selected based on a limitation of our load generator, since it could not be operating stable at the minimum IP packet size of 64 bytes. It was also decided to use UDP packets for the measurements, as they are sent at a constant rate and no congestion control or flow control algorithms like in TCP could change our load level. Finally, the interval of 15 steps was selected, because it allows to minimize the amount of tested packet sizes, while still maintaining a decent spectrum of packet sizes.

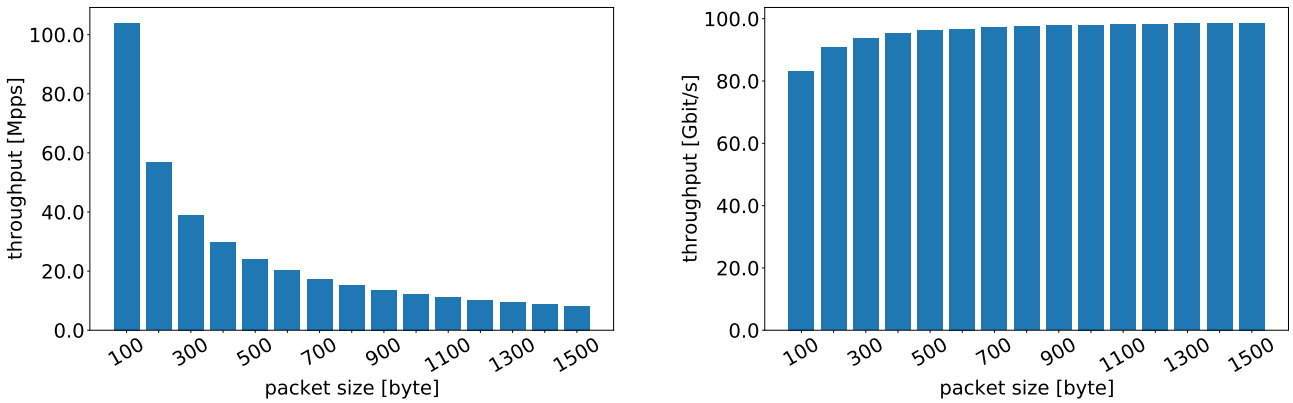
Within this test the lower packet sizes are the most challenging ones for the hardware, because small packet sizes will increase the number of packets per second that can be transmitted before saturating the link bandwidth. For example, at a 10 Gbit/s link speed it is possible to transmit around 12,5 Million packets per second, at a packet size of 100 bytes per packet. In comparison at 1500 byte per packet, it is only possible to forward around 830 Thousand packets per second on the same link. This means at lower packet sizes the hardware must handle many more individual packets, compared to fewer ones with larger sizes. However, this effect can be seen later on at the forwarder in more detail. In order to generate a 100 Gbit/s traffic stream, the duplication feature of P4STA is used, since Iperf3 can't generate such a high-capacity flow. A duplication factor of 150 was selected in the P4STA load generator.

When the cache is tested it is expected to handle full line rate at 100 Gbit/s, because its hardware was designed for this use-case. Additional to the forwarding performance in terms of pps and bps, we will also measure the latency. The latency of each packet will allow us to evaluate if the device performance is stable, even in overload situations. This can be evaluated based on the assumption that if a device performs stable forwarding, the latency should be measured in a small window of variance. If the device should have issues in different situations, the latency could become unstable and for example introduce latency jitter and a generally increased latency. One specific example could be a higher than expected latency that has a stable upper limit, which can be an indicator that the device is overloaded and its buffers are full. The time the packets are stored in the buffers and waiting for the processing is measured in the additional latency. Finally, it is also important to track the packet loss, to see if any packets are lost at the two tested devices.

As the cache Tofino is an 100 Gbit/s device, we have set up P4STA to perform 15 tests at the 15 different packet sizes. To ensure the packets are forwarded by the cache any time during testing, we have configured the controller to install the routes for the load generator permanently into the cache. We have also verified this behavior based on the cache's interface counters.

As we tested this first, we experienced some issues at the P4STA load generator when we have saturated the 100 Gbit/s port to P4HC to 100 percent capacity. We detected that buffer issues happen on the duplication of the packets at the Tofino of P4STA, so that we essentially did not receive any valid measurements. To overcome this issue, we configured a hardware shaper to 99900 Mbit/s which is almost 100 Gbit/s and we started to receive valid measurements. We have then performed a 10-seconds-long test run per packet size and evaluated the results. As we started to compute the metrics for pps and bps, we have again faced some issues. We had realized that P4STA was designed to measure latency in first place instead of bandwidth, so P4STA itself was not able to measure the packets per second natively. However, P4STA can measure the exact number of packets that have been passed to the cache in the 10 seconds duration. So, we computed the average packets per second based on the number of packets in the 10 seconds test run. Based on the same principle we have computed the average bits per second, based on the packets of 10 seconds and the known packet size.

In Figure 6.2 the results of the test can be seen. On the left in (a) the packets per seconds per packet size can be seen and on the right in (b) the bits per second are presented. At the packets per second graph, it can be seen than we were able to forward around 104 Mpps at 100 byte packet size with our



(a) Cache throughput in packets per second.

(b) Cache throughput in bits per second.

Figure 6.2: Throughput of the Tofino cache.

cache. While this is quite a good link utilization, we realized that the line rate for 100 byte packets at 100 Gbit/s link speed should be around 125 Mpps. On a closer look we have noticed that P4STA was not able to generate that number of packets to reach line rate. So, the cache is performing well at low packet sizes. When the packet size is increased, it can be observed that the packets per second are decreasing, just as expected. In the Figure 6.2 (b) it now can be observed how the throughput reaches the exact line rate of 100 Gbit/s. From this point of view the performance meets our expectations, so we have started to evaluate the latency.

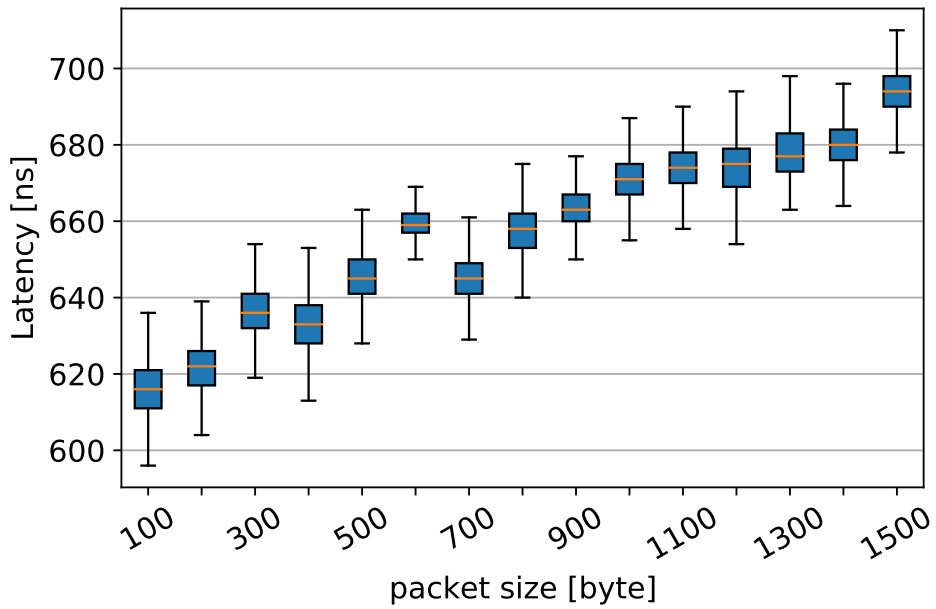
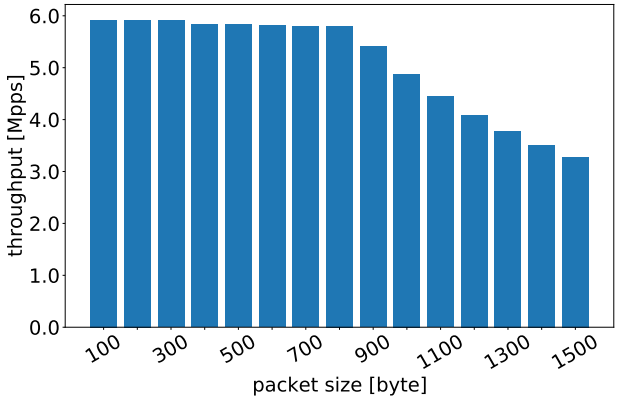
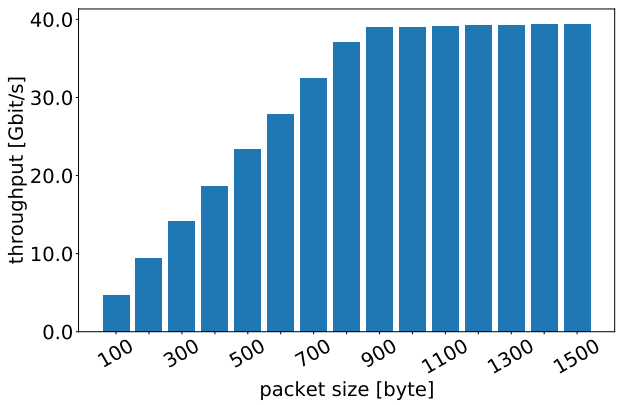


Figure 6.3: Latency of the Tofino cache.

During the test runs we have sampled every 100th timestamped packet to the external host. We have then computed the latency and have generated a box plot of the latency for each packet size. We have some outliers among the latencies that were omitted, since they were not within relevant areas. The results can be seen in Figure 6.3. The figure shows that the latency was stable over all packet sizes and the most measured latencies are within a close range in one packet size. However, it can also be seen



(a) Overloaded forwarder throughput in packets per second.



(b) Overloaded forwarder throughput in bits per second.

Figure 6.4: Throughput of the DPDK forwarder with a load larger that 40 Gbit/s, resulting in a overload situation.

that the latency is increasing with the packet size. This can be explained by the longer packets itself, since sending and receiving more bytes takes longer.

All in all, the performance of the cache is as expected. As far as we could test, the Tofino cache reaches line rate performance in most situations. This was expected due to the cache running on the Tofino switch, which is designed to deliver line-rate performance and has been proven by several related work papers before [KNB+20]. This result proves that P4HC is able to deliver line rate performance for cached entries. Also, the resulted latency shows the minimum latency (around 660 nanoseconds on average) that is introduced by P4HC to a packet, since it is the fastest path in P4HC.

6.3 P4HC DPDK-Forwarder Evaluation

In the last section the caches performance was evaluated. In this section the forwarder performance will be evaluated. The same performance tests as before will be used at the cache. 15 measurements from 100 bytes per packet to 1500 bytes per packet with the increase of 100 bytes between measurements are performed. Each test is again performed for a duration of 10 seconds with a duplication factor of 19, which means that the IPerf load will be increased by a factor of 20. The duplication factor was selected accordingly to overload the forwarder, as the forwarder is connected with a 40 Gbit/s link to the cache. Within all tests, it must be ensured that more traffic is generated than the 40 Gbit/s link can handle, in order to overload the forwarder and measure its maximum performance. Afterwards only packets are measured and sampled for the latency if they have passed the forwarder. Before the results are provided it should be noticed that it is not expected that the forwarder will archive line rate at small packet sizes. This assumption is based on the current state of forwarder-implementation in the DPDK application for now. To ensure that no routes are inserted to the cache during the experiment, the offloader was disabled at the controller.

The result of the measurements can be seen in Figure 6.4. In (a) again the packets per seconds are shown and in (b) the bit per second. At the first look in Figure 6.4 (a), it can be seen that the forwarder has a hard limit at around 5.9 Mpps. The line rate at 40 Gbit/s would be around 50 Mpps for this particular packet size. The limitation of the 5.9 Mpps can be observed up to a packet size of 800 bytes. Afterwards at 900 bytes, the forwarder reaches around 5.4 Mpps, which is very close to the 5.5 Mpps that would be line rate at 900 bytes. This can be confirmed within Figure 6.4 (b) where it can be observed that at 900 bytes packet size a throughput of nearly 40 Gbit/s is reached. The difference of 0.1 Mpps can be led back to measurement inaccuracy, when the pps values are calculated based on all packets that were transmitted during the 10 seconds, as it is an average. If there is a slow start for example, so

that the rate has not reached the 100 percent load from the first moment on, this will impact the rate negatively on the average later. In both graphs in Figure 6.4 it can be observed that from 900 byte to 1500 byte the forwarder is operating at line rate.

As it was detected in the first analysis, the forwarder's limitation is at around 5.9 Mpps. This behavior was expected and can be explained by the implementation of the forwarder. While implementing there was no primary focus on optimizing the DPDK application. So, the forwarder operates only at one CPU core. However, 5.9 Mpps for one CPU core is still impressive and proves the possibilities, if the application would be optimized further for multithreaded forwarding. Before the next measurement is provided, a look into the latency is taken.

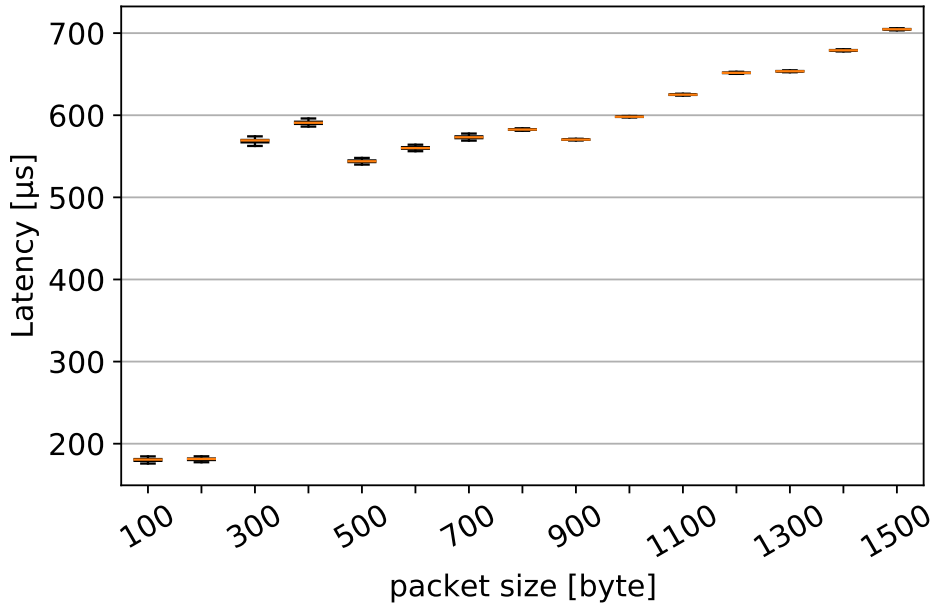
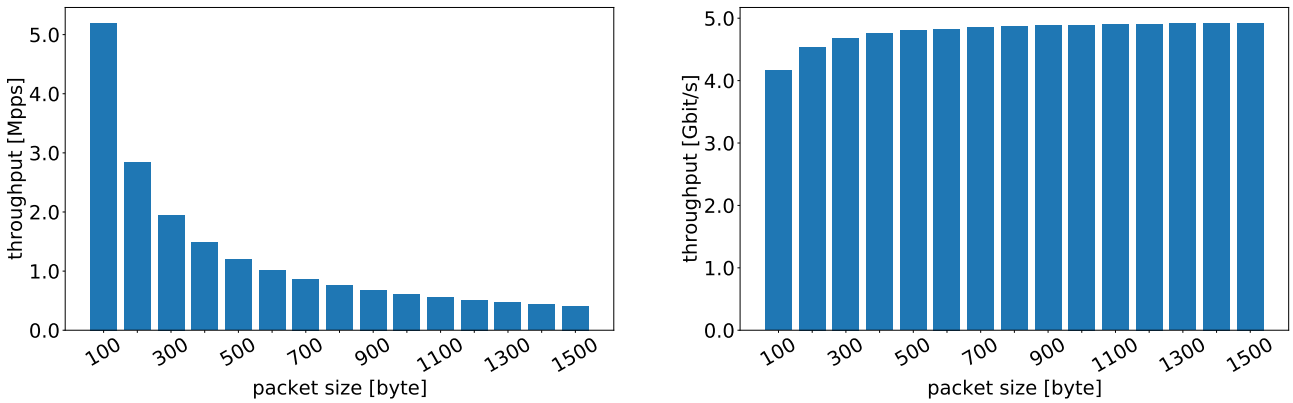


Figure 6.5: Latency of the DPDK forwarder with a load larger than 40 Gbit/s, resulting in a overload situation.

Figure 6.5 shows the latency of the test run. The box plot is built again like before at the cache, by analyzing every 100th timestamped packet and the outliers are cut off. It can be observed that the latency is almost constant until 900 bytes, where the line rate was reached. Afterwards the latency rises in a linear fashion according to the packet size. However, the first two sizes, 100 and 200 bytes, are clearly standing out with latencies below 200 microseconds. At this point we are not able to explain the behavior at 100 and 200 bytes. The measurements have been performed 10 times with the same result. Interestingly, the latency is still higher than the latency values when the forwarder is not overloaded, as it will be shown in the next section. At this point it should be referred to future work to investigate this behavior in detail. However, the behavior of the latency after 200 bytes is still interesting, as the latency is nearly stable until reaching 900 bytes. This can be tracked back to full buffers within the DPDK forwarder application. This also explains the overall high latency compared to the non-overload latencies, which will be provided afterwards. At this point it also can be proven that the overloading of the application is stable, since the latency does not show any high scattering. Again, there were some outlier measurements omitted which did not affect the outcome.

In the previous section, the forwarder's performance was measured in an overload-scenario. The forwarder will now be tested again within normal non-overloading conditions. In the measurement before, the limit of 5.9 Mpps was ascertained. To investigate the forwarder's behavior at normal conditions, a load must be applied that is less than the breaking point. It was decided to configure a shaper at P4STA to 4.9 Gbit/s per second, which should protect the forwarder from overloading. With this configured shaper, the performance test from before was repeated with 15 steps from 100 byte to 1500 byte packet



(a) Forwarder throughput in packets per second.

(b) Forwarder throughput in bits per Second.

Figure 6.6: Throughput of the DPDK forwarder on non overloading conditions. (Load under 5 Gbit/s.)

size. For each step the test was performed for 10 seconds, while every 100th packet was sampled to measure the latency. And again, in the latency graph the outliers have been removed.

In Figure 6.6 again the results from the test on packets per second and bits per second can be seen. It can be observed that the load only hits around a maximum of 5.1 Mpps at peak, which proves that the forwarder was not overloading. Also, it can be observed that the throughput in Figure 6.6 (b) is close to the configured shaper of 4.9 Gbit/s. However, the more interesting results are the latencies, which are presented in Figure 6.7. It can be seen that the latency in the non-overloading condition has fallen rapidly from around 520 – 710 microseconds to 5 – 11 nanoseconds. This decrease of latency is quite significant in comparison to the overload conditions. Also, it must be noticed that the latency from the forwarder path includes the latency of passing the cache two times. After being generated on the load generator, packets will then traverse the cache towards the forwarder, where the routing decision is actually made, and are then sent back through the cache to the load generator again. So to get an idea of the latency of the forwarder application itself, the average latency of two times 650 nanoseconds can be subtracted from the latency of this measurement. If the measurement of 100 byte packets is ignored in Figure 6.7, the average latency is around 7.5 microseconds. So, if two times 650 nanoseconds for the cache are subtracted, the average latency of the forwarder application is around 6.3 microseconds after all. This seems to be a really good value, compared to the cache. In comparison the software data plane is only around 10 times slower than actual switching hardware, in terms of latency. However, in the measurements it can be seen that the latency of 100 byte packages is quite a bit higher than the average. The increased latency can be explained by the much higher packet rate at 100 bytes sizes. At 100 byte packets, the load was 5.1 Mpps and on 200 bytes it was just 2.8 Mpps, so a total difference of 2.3 Mpps. This is almost double the amount. However, in comparison the latency has not doubled.

The last two measurement have shown the worst-case and the best-case performance of the P4HC forwarder. Even if the values are astonishing for one CPU core, we known that it should be possible to go even further. Not in terms of lower latency, but more in terms of pps at lower packet sizes. Our measurements have proven that one CPU core can enable up to 5.9 Mpps of forwarding capacity, so if more cores are used for the forwarding, the performance should scale almost linear on the number of cores. To prove this, we have quickly taken a look into the DPDK documentation and found a feature called Received side scaling (RSS) [dpda]. RSS allows to distribute the incoming packets at the NIC to multiple queues. This allows to assign one CPU core to one receive queue on the NIC. The distribution of the packets to the queues is performed by a hash function at the NIC. When a packet is received, it will be hashed and mapped to one queue on the NIC, based on the hash-result. Afterwards one CPU core is assigned to each queue to process all packets. This principle can be seen in Figure 6.8.

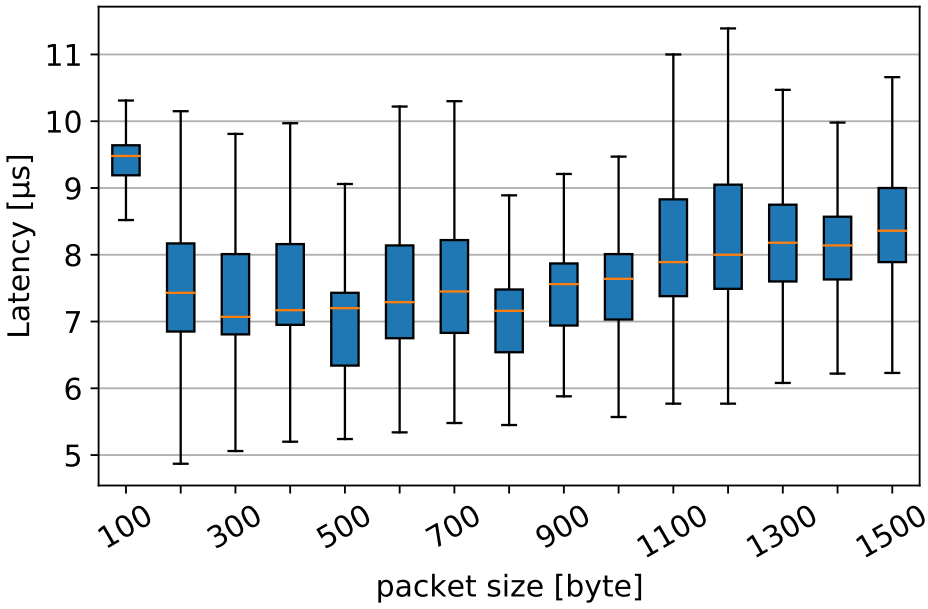


Figure 6.7: Latency of the DPDK forwarder under normal conditions. (Load under 5 Gbit/s.)

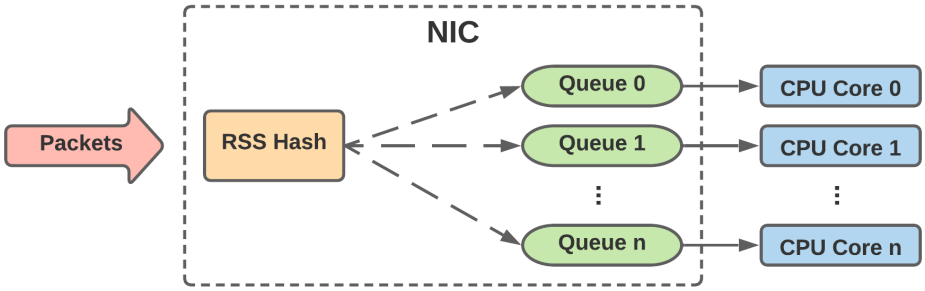
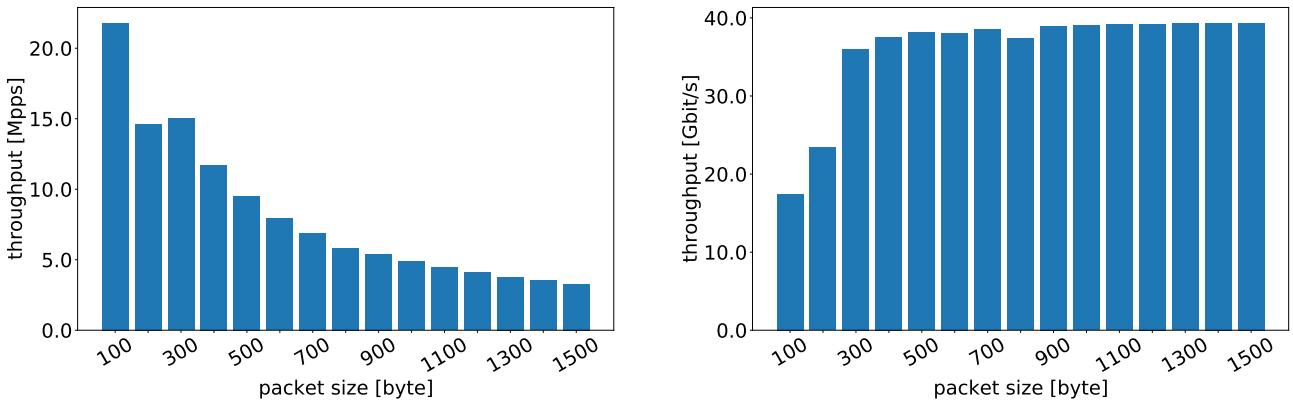


Figure 6.8: Receive Side Scaling (RSS) at the NIC level.

The support of multiple queues was implemented and it was decided to use 8 CPU cores for forwarding. Afterwards the same test as before was executed. At this point it was noticed that the forwarder application did not scale as expected linear to the core count. It has quickly shown that the scaling depends mainly on the capability of distributing the packets to the individual queues of the NIC. As it was found out, this works at best if many different flows are sent through the application, to enable the hashing at the NIC to distribute the packets more evenly. It has been shown that one flow is mapped to one queue of the NIC. This means it would be required to generate more flows with our load generator. As it points out, P4STA works per default with three flows in parallel. So, we were able to achieve three times more of the performance as before. It was also tried to increase the flow count of the load generator, however the issue here was that due to the duplication and the design of the load generator it was not possible to perform any repeatable stable test run with more than three flows. In some cases it was possible to achieve 30 Mpps at 6 flows, but the issue was that the load generator was not stable enough to perform our full 15 pass test without errors. So it was decided to present the result of 8 CPU cores with 3 flows, which could be tested stable.

The result can be seen in Figure 6.9. In (a) again the pps are provided and in (b) the bps. In the graph (a) can be seen that it was possible to achieve 21,8 Mpps at 100 bytes. So, in this case the forwarder has scaled linear in terms of flows. Before at one core, the forwarder has achieved 5.9 Mpps



(a) Forwarder throughput in packets per second.

(b) Forwarder throughput in bits per second.

Figure 6.9: Throughput of the DPDK forwarder at 8 CPU cores and a load of 3 flows.

for three flows. Now the forwarder performs more than 3 times better. It also shows that the main load is distributed only to three cores, which limits the performance again. In the latency graph which is presented in Figure 6.10, it can be seen that the cores which are processing the packets are at their limits, as the latency equals the values to the overloaded case from the beginning of the chapter (in the box plot again the outliers were removed). Also, the increased spread of the latency indicates that the cores are performing differently and at 500 and 1000 bytes, it can be observed that some cores outperform others. In this case it can be expected that some of the cores are not receiving as many packets as the others, so that their overload is much lower. Another interesting finding is the measurement of the 200 bytes packets. Here again, as in the first measurement of the chapter, the core shows a latency below 200 microseconds, which is the same unexplainable behavior as before. Also, this behavior should be investigated in future work, since this measurement was repeatable multiple times. However, this small code adjustment shows what is possible with the forwarder performance, if the code is further optimized and also what is possible with software routers in general. It would be interesting to find out whether the performance could be increased even more by code optimizations, or the testing of the forwarder with another load generator which can generate many flows. This however could also be a topic for future work.

In this section all facets of the forward performance were examined. The performance and the latency and overload conditions were investigated, which shows the worst performance cases of P4HC and the best case in terms of latency. Finally, based on some quick optimizations, it was shown how the forwarder's performance could be scaled out under the right conditions.

6.4 P4HC Offloading Evaluation

In the last two sections the cache's and the forwarder's data planes were evaluated individually. Since P4HC is a hybrid router switch, now both data planes must also be tested together. In order to do so, P4HC was fully set up with both data planes and the controller with the offloader enabled. In this section the transition of a route from the forwarder to the cache is evaluated. To evaluate the transition, the test plan must be adjusted. Instead of running 15 different load types for one test only, one load per test will be applied. The focus will be put on two situations. The first situation is when the forwarder is not overloaded and the route is installed to the cache. The second situation is when the forwarder is overloaded and the route is installed into the cache.

For both situations we created a specific load and apply it for 10 seconds with P4STA. Within the 10 seconds, the controller must be able to detect the new flow and offload it. As it was introduced, the offloader and the metrics collector are working at a 10 second time interval each. Since with these

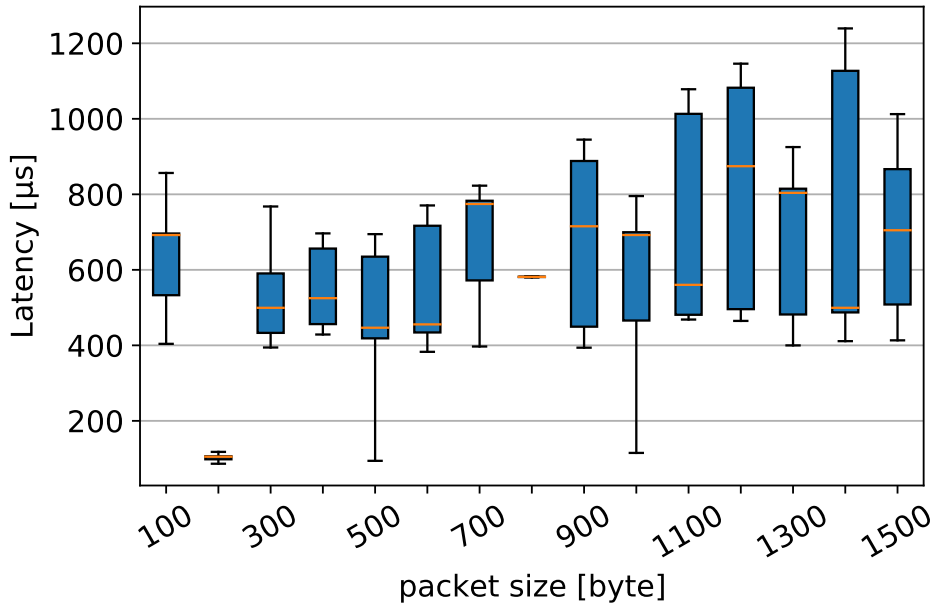


Figure 6.10: Latency of the DPDK forwarder at 8 CPU cores and a load of 3 flows.

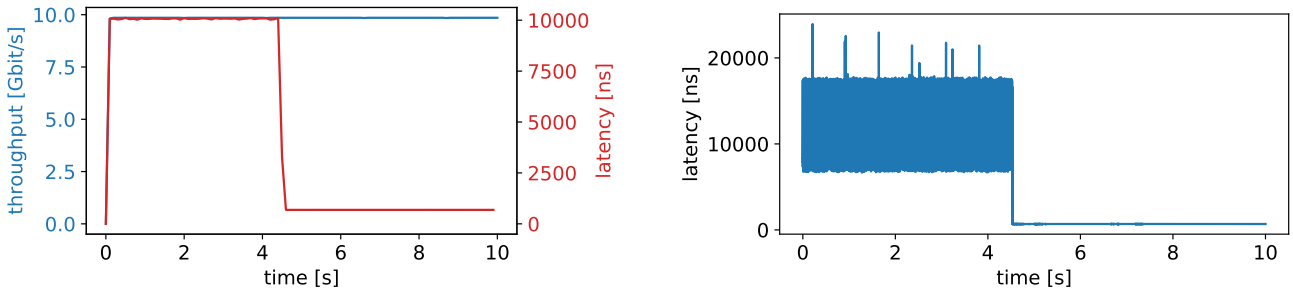
intervals the offloading decision could take longer than 10 seconds, the update intervals were adjusted at the metrics collector to two seconds and at the offloader to three seconds. It should be noticed that with the short interval the metrics collector could deliver inaccurate data. As mentioned in the design, shorter sampling intervals could lead to the issue that the reported bandwidth is too low, because too few samples have been captured. However, in this test it is sufficient to detect the flow regardingless of its bandwidth, since there are no other flows to interfere.

For each test two graphs will be generated. One that includes throughput and the latency within one plot. In this graph it is possible to study whether the offloading decision affects throughput and the latency at first and also show the point at which the offloading is happening. The latency should drop from the observed forwarder latencies down to the observed cache latencies immediately on offloading. To remove jitter in the latency graph, the average latency is computed for every 100 milliseconds. As this down-sampling in the plot could hide some detail regarding the latency measurements, it was decided to plot the latency in an additional graph, to investigate if something unexpected happens. During the test again every 100th timestamped packet is sampled, in order to generate the latency graphs. Also, the sampled packets are used to compute the throughput over time, because as mentioned above, P4STA does not measure the throughput per second, it will instead measure the throughput of the complete test only. As before only the average throughput per second was required, which is easy to compute. So now another solution was required. It was decided to calculate the effective throughput at each second, based on the sampled timestamp packets. As it is known that every 100th packet is sampled, the throughput of the sampled packets is calculated and multiplied by 100. Finally all sampled packets are examined for out of order packets. This phenomenon will be explained later on, when it is detected.

In the next two sections the investigation of the two experiments is described in detail. Depending on the result it, will be decided how to proceed afterwards.

6.4.1 Normal Operation

In the following, the normal operation of P4HC is investigated, which means a situation where a route is offloaded to the cache under non-overloaded conditions. Most important for this test is that the forwarder is not overloaded. To ensure the forwarder is not overloaded, the load generator is configured



(a) Throughput vs Latency.

(b) Latency during the offloading process.

Figure 6.11: Offloading process in a non-overloaded situation.

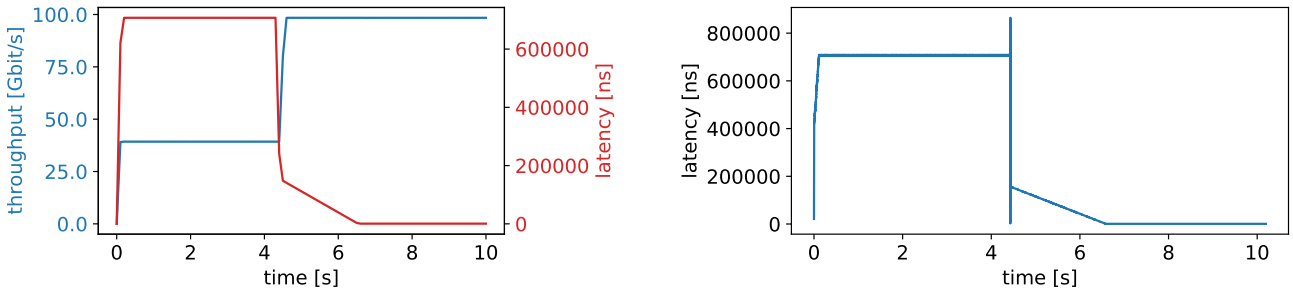
at a rate limit of 10 Gbit/s. As a load, UDP packets with 1500 bytes length were selected, based on the previous learnings, because the forwarder struggles to keep up at small packets with high packet rates. 10 Gbit/s is far from the limitations of the forwarder, as it was capable of line rate performance at 40 Gbit/s at 1500 bytes packet size.

Figure 6.11 shows the results of one of 10 test runs that were made. In (a) the transition from the forwarder to the cache can be observed. As the test starts, it can be seen that the load is applied as expected with 10 Gbit/s and the latency is on average at around 10 microseconds. The latency in the beginning is also within the expected range of the previous test of the forwarder, so it can be assumed that the forwarder is performing as expected without any overload. After second 4 it can be observed that the latency will drop down to around 685 nanoseconds. At this point the controller has performed the offloading. While the offloading process was performed, it can be seen that the actual throughput remains stable. That means that the transition happens without any negative impact to the flows. Also, P4STA did not report any packet loss during the test. If the figure (a) is investigated closer, it can been seen that the latency did not go straight down to 685 nanoseconds. Before reaching the 685 nanoseconds, there was one measurement in between which is responsible for the short delay in the down falling graph. It can be assumed that the bend is generated while calculating the average for every 100 milliseconds, so that in this timeframe there are packets considered from before and from after the offloading process. This behavior can be confirmed in figure (b). The graph shows all latency measurements during the test. At the offloading time point, the graph directly falls to the cache's latency. Interestingly, one can observe a higher jitter from the forwarder compared to the cache. However, this can't be examined in detail within this graph due to scaling problems, since the forwarder operates at microsecond latency and the cache at nanoseconds, so the jitter couldn't be seen. Since the forwarder introduced jitter around 10 microseconds, it can be confirmed that the jitter is very low in comparison.

Within the plotted test, no out of order packets have been detected. However, that does not mean that there were none, because only every 100th timestamped packets were sampled. Which could lead to the problem of missing those packets. The test was performed 10 times and in none of them any out of order packets have been found. Also, the behavior of the offloading in all 10 tests matches the one that was presented. The only difference was the point in time when the offloading was performed. However this is expected, since the response time of the offload could vary, depending on the timings between the start of the test, the interval of the collector and the offloader itself.

6.4.2 Overloaded Operation

In the last section the offloading transition under normal conditions was investigated. In this section, the case where the forwarder is overloaded at first and then the load reduced by offloading is investigated. As load for this test, it was decided to generate 100 Gbit/s of UDP traffic at 1500 byte packets. This situation should allow the forwarder to forward its full 40 Gbit/s of capacity, while being overloaded.



(a) Throughput vs Latency.

(b) Latency during the offloading process.

Figure 6.12: Offloading process in an overloaded situation.

After offloading, the cache should be able to handle the full 100 Gbit/s. As mentioned before, due to some side effects when pushing the load generator to its 100 percent link speed, a shaper was configured to 99,99 Gbit/s. Within this test also every 100th timestamped packet is sampled, to measure the latency and evaluate indications for out of order packets.

Figure 6.12 shows the result of one test run. In (a) again the throughput and the average latency are provided. In (b) all latency measurements are provided. At the first look both graphs differ from the previous test. At the beginning in graph (a), it can be observed that the throughput at the forwarder is around 40 Gbit/s, as expected. The latency on the other hand is at 700 microsecond, which is significantly higher than in the non-overloaded situation before. However, the latency of 700 microseconds at overloading condition covers with the values of the previous test of the forwarder on overloading conditions. Shortly after 4 seconds, it can observed that the controller has performed the offloading process. After the offloading process has happened, the throughput increases up to the expected 100 Gbit/s. The interesting part in this case is the average latency: it can be observed that the latency is dropping rapidly to around 90 microseconds and then slowly starts to decrease linear, down to the expected cache's latency at around 690 nanoseconds. This first fast decrease and the afterwards linear down falling of the latency differs significantly to the previous situation. To get further insights, the latency graph in (b) should be investigated. In the graph it can be observed that at first the latency, is as expected, at around 700 microseconds in the beginning and around 690 nanoseconds in the end. Within the transition zone after 4 seconds, the transition can be observed. At first it could be seen that when the offloading happens, the latency increases and decreases at the same time. This indicates that some packets directly take the cache's direct path, resulting in the cache's latency of around 690 nanoseconds. The increased latency which is also measured could indicate a buffer that is getting filled. An idea would be the output buffer of the cache's Tofino switch. As the forwarder is in an overloading state, its buffers should be full and the egress buffers on cache at the link to the forwarder, too. As the offloading occurs, the packets from the load generator are directly sent back to the packet generator at 100 Gbit/s. Additionally, there are still packets left in the buffers from the forwarder and its switchports. Those packets are now also waiting to be forwarded back to the load generator. In our opinion, this will overload the 100 Gbit/s return link from P4HC to the load generator and the egress buffer on this port is filling up. The filling of the egress buffer will add additional latency to the packets from the forwarder. As the load generator is continuously sending 100 Gbit/s, the buffer can only empty slowly. It also could be happening that when the egress buffer is full, also the ingress buffer of the switch starts storing packets from the load generator. This explains the linear drop after the up and down peaks of the latency. After the buffer is empty at around 6.5 seconds, the latency has normalized at the cache's expected values. This behavior should be investigated further, to check if our hypothesis can explain the described behavior.

Also, this behavior and, as well as our hypothesis, implies that there should be out of order packets. Before this is investigated, it will be shortly explained what out of order packets are. The load generator sends the packets one after another in a constant order. As the packets hit P4HC, there are two possible

paths with different latency each. A packet trough the forwarder path normally requires around 700 microseconds to be forwarded by P4HC. On the other hand, there is the cache's path in P4HC, where a packet only requires around 700 nanoseconds. When now two packets are sent in order from the load generator and the first one is sent through the forwarder path and the second one is forwarded through the cache, because the route was offloaded in between those packets, the packet that was sent at last would be arriving before the first one, due to taking the path of lower latency. This behavior can also be observed in the Figure 6.13.

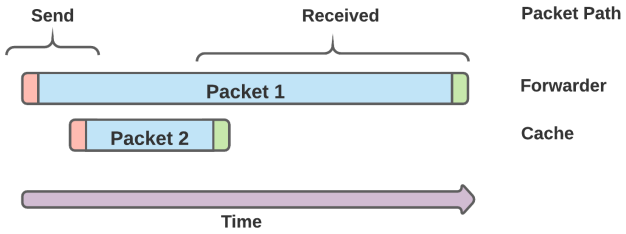


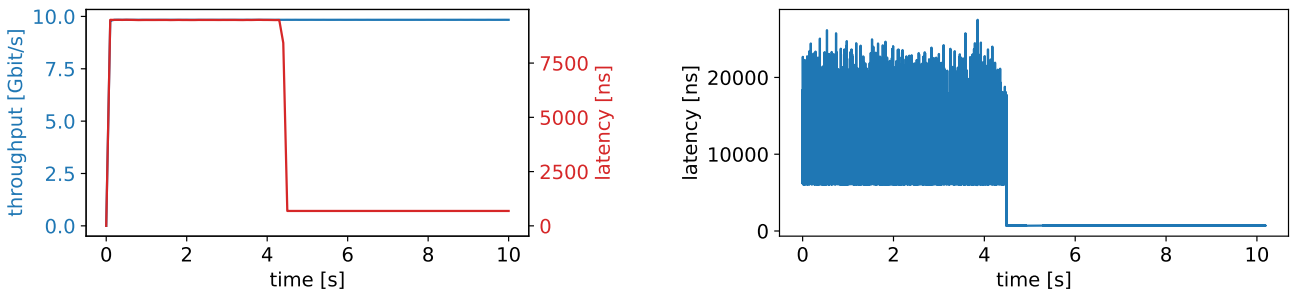
Figure 6.13: Out of order packets during offloading in P4HC.

In the figure is shown that the first packet is received by P4HC before the second, but is sent back to the load generator after the second one. To confirm this behavior the result of this test was investigated to find some out of order packets, which were actually found. To ensure that the described behavior is not only a single run phenomenon, the test was repeated 10 times. In each run the exact same behavior could be observed, including out of order packets. As P4STA was not able to capture every timestamp at 100G and only every 100th packet was measured, it can be assumed that not all out of order packets have been found so far. Also, after explaining the out of order problem, it should be obvious that the problem must also be present in the non-overloaded case. This behavior should be investigated further in the next section.

6.4.3 Detailed measurements

In the last section it was shown that the sampling of the timestamped packets leads to the issues that not all out of order events can be observed. Or in the case of non-overload, we did not see any out of order packets at all. In the next section, the experiment from the section above is repeated. This time every timestamped packet is sampled and analyzed. In order to evaluate that many packets, a more powerful external host for the packet capturing is required. The standard external host of P4STA can not handle that many packets, since it is written in Python. For other uses cases, P4STA provides an external host based on DPDK to capture every packet. At the time of writing, only a 10 Gbit/s NIC was available at the external host, so the experiment from above could not be repeated exactly, as the NIC of the external host does not provide the required bandwidth. The first experiment under normal condition can be performed as before at 10 Gbit/s. However, the second experiment uses 100 Gbit/s and can't be captured from the external host. It was decided to overload the forwarder this time based on the packet rate, which allows to overload the forwarder before saturating the 10 Gbit/s link bandwidth.

In the first experiment the load generator was configured to generate 10 Gbit/s of UDP packets at 1500 bytes packet size. The test was executed again for 10 seconds, while every packet was captured. The results can be examined in Figure 6.14. In (a) again the throughput and the average latency is presented and in (b) all available latency measurements are shown. At first, in graph (a) it can be observed that the load generator generates exactly 10 Gbit/s over the complete test. Shortly after second four, the offloader has performed the offloading. And the average lantency has fallen from around 9.5 microseconds down to 686 nanoseconds. Those averages cover with the initial measurements of the cache and forwarder and thereby confirmed that the forwarder was not overloaded during the measurement. In graph (b)



(a) Throughput vs Latency.

(b) Latency during the offloading process.

Figure 6.14: Offloading process in a non-overloaded situation while capturing every timestamped packet.

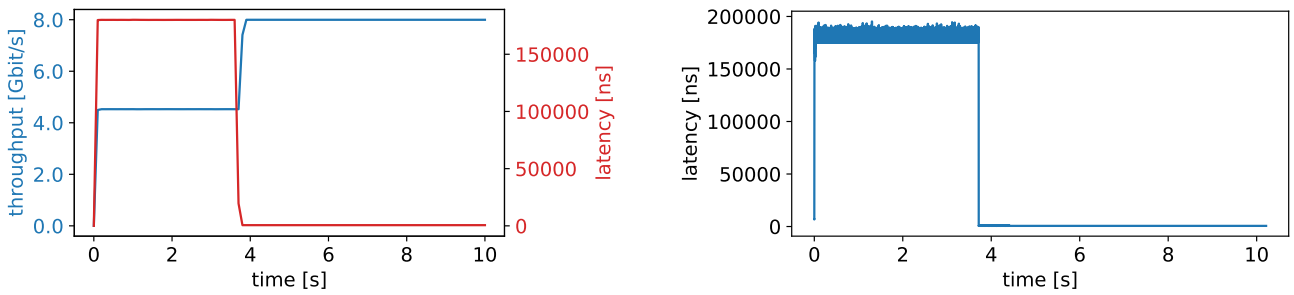
all latency measurements are plotted. Here can be observed that at the forwarder there is an actual hard limit for the minimal latency at around 6 microseconds. Over 17.5 microseconds are not many measurements to be found, as the graph also begins to fade out. The average of the latency of 9.5 microseconds from graph (a) also confirms this assumption. As the result looks mostly the same to the previous measurements, now the out of order packets are investigated. And again, in the presented run, from the graph, we could not find any out of order packets. We performed 10 times the same test and have found out of order packets in 7 runs, while in 3 runs we did not find any. Table 6.1 presents the number of out of order packets per test.

Experiment	1	2	3	4	5	6	7	8	9	10
Out of order packets	1	0	1	0	1	3	4	1	1	0

Table 6.1: Out of order packets during offloading of the 10 runs.

This confirms the assumption that also in normal operation there should be out of order packets. However, the question is why there are three tests that did not show any out of order packets. One explanation would be that the 10 Gbit/s shaper at P4STA sends the packets in bulks, so that there are time frames where no packets are traversing through P4HC and this timeframe is equal or larger to the latency of the forwarder. So, if the offloading timing is right and the offloading happens in the frame where no packets are transmitted. A good indicator for this is also that when out of order packets are found, they only occur at low amounts. Also it can be noticed that in the tests only a few packet are sent in comparison to the capabilities of the 100 Gbit/s link capacity. For example in the test with 1500 bytes at 10 Gbit/s, only 0.8 Mpps are transmitted. Compared to the possible 8.3 Mpps at 100 Gbit/s. Which means, if the packets are distributed evenly, every 1.2 microseconds one packet would be transmitted. If the shaper did not send the packets equally, larger gaps where no pakets are transmitted can occur. For example, if the packets are sent in bulks of ten, there would be a delay of 12 microseconds between the two bulks, which is a time frame that is larger than the latency of the forwarder. If now for example the offloading happens directly after sending one bulk, all 10 packets are sent back trough the forwarder in around 9.5 microseconds, so there is still a delay left of 2.5 microseconds where no packet is sent and the new route is installed. So in this case there is no packet that can traverse P4HC faster than the previous ones. When the offloading happens during sending of a burst, out of order packet are occurring, because the some packets are getting forwarded with the cache latency. So, it seems to be that in this test case absence of out of order packets can only happen in some cases. The results of the ten tests are confirming this.

As in the non-overloaded situation, it was not guaranteed to see out of order packets. However, in the overloaded case it seems to be more common to see the out of order packets. As mentioned before in this experiment, it is only possible to capture 10 Gbit/s of timestamped packets at the external host. So



(a) Throughput vs Latency.

(b) Latency during the offloading process.

Figure 6.15: Offloading process in a overloaded situation while capturing every timestamped packet.

the load must be selected to overload the forwarder, while not exceeding 10 Gbit/s of bandwidth. From the performance analysis of the forwarder, it is known that the offloader has its limit at around 5.9 Mpps with 100 byte packets. The resulting throughput is around 5.9 Gbit/s, which can be captured by the external host. A rate limit was configured to 9.9 Gbit/s and the load was set to 100 byte UDP packets. This will generate more packets than the forwarder can handle, while it protects the external host from overloading the 10 Gbit/s link.

The result of a test run can be seen in Figure 6.15. In (a) again the throughput and the average latency is provided and in (b) all available latency measurements. In the beginning of (a) it can be observed that the packet rate is a bit lower than expected at only 4.5 Gbit/s. The latency is around 175 microseconds. This is interesting, as it was observed before that the latency on smaller packets is lower, even in overloaded conditions. After second 3 the offloading was performed, which has the consequence that the latency decreases to the cache's latency and the throughput has reached 8 Gbit/s. At this point the behavior of the offloading seems to be equal to the non-overloaded condition before. The latency in (b) confirms the same behavior. That was not as expected, as the intention of the overloading test was to reproduce the overloading situation from the test with 100 Gbit/s in the last section. However, the behavior could not be replicated at lower rates. This also shows that the initial theory is still valid and the behavior occurs only when the buffers of P4HC are starting up to fill and then free up slowly over time. So the investigation of this behavior must be delegated to future work, since at the time we did not have the lab equipment to measure the latency behavior during the offloading at 100 Gbit/s.

Finally, again the out of order packets were measured. In the presented test run 1071 out of order packet have been found. This hardens the theory from before that at higher packet counts it will be more likely to see more out of order packets. However, it should be noticed that the out of order packets are only an interesting side effect of the offloading in P4HC. In general, out of order packets should be no problem, since networks generally do not guarantee in order delivery at layer 3. On the Internet out of order packets can occur at any time, as well as packet loss. It is the task of the upper layer protocols, such as TCP, to deal with the reordering of the packets.

6.5 P4HC Control Path Evaluation

In the last sections the forwarding performance of the data planes of P4HC was evaluated. As mentioned in the introduction, there is one performance attribute of the data plane which is shared with the control plane. How fast the data planes' table entries can be updated. The performance for manipulating the data planes' entries can quickly become important when the controller needs to modify a large set of routes in one of the data planes. A benchmark was designed to evaluate the performance .

Since the routing entries are the most important entries in the data planes, the benchmark will be designed for insert and delete operations. It was decided to measure the insertion and deletion rate in operations per second. Also, it was decided to measure the time that is consumed by the call of

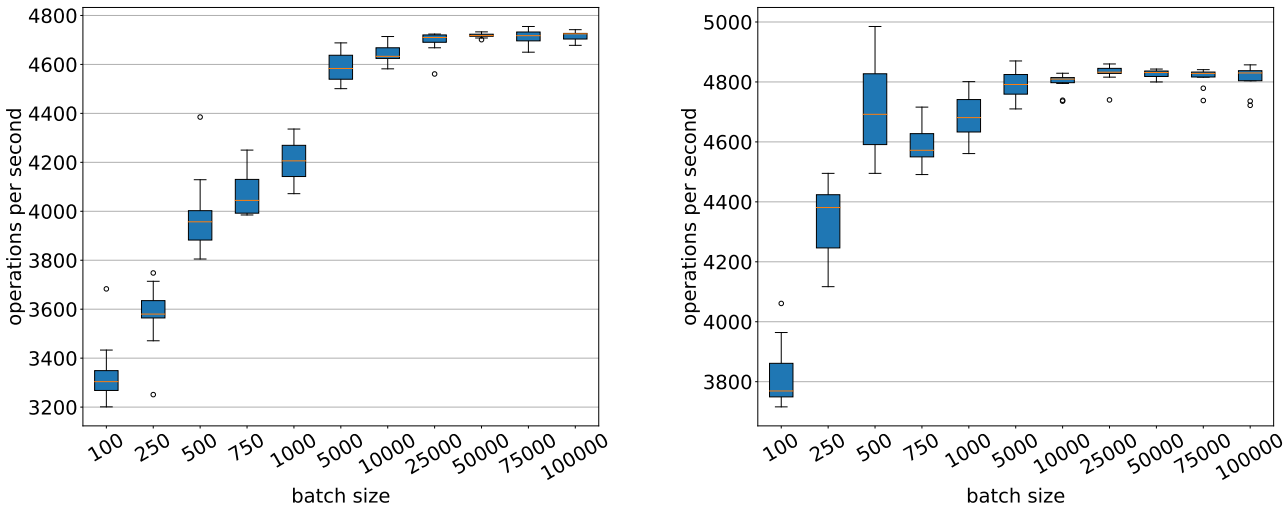
an update operation as install duration. These two measurements are important for P4HC due to the following reasons: first, the update rates of the data plane determine how many routes can be learned from BGP and installed into hardware. Second, how long does it take from the offloading decision to the actual insertion or removal in hardware. And finally, depending on how much updates can be performed, is there an optimum number of changes that can be performed during an offloader iteration.

The key point of this measurement is the controller's view, so the benchmark was implanted within the controller. Based on a configuration flag, the controller will load the benchmark after the data planes are initialized, but will neither start RIB, offloader or routing daemon afterwards. To measure the operations per second, the benchmark will insert *n* entries into one data plane and remove them afterwards. For each insert and delete, the duration of the method call is measured in Go with nanosecond precision. The invoked methods are the ones that were defined by the cache and forwarder interface, so the benchmark should run with any future data plane too. Also this allows to include the overhead of preparation of the update messages for the data planes, as well as the transmission times for the updates and finally also the time the data plane requires to update. This allows to measure the effective rate at which the controller could be utilized, including the real time period that is required for one update. This decision is quite important, because if the data planes update rates would be measured outside of the controller, it could be possible that the performance differs. It was also decided to run the benchmark in batches with different batch sizes, to investigate if the update rate or the insertion delay will change when the tables are getting full or if there are any side effects on continuous updates. The following batch sizes were selected: 100, 250, 500, 1.000, 5.000, 25.000, 75.000, 100.000. It was decided to stop at 100.000 entries, because the cache could not handle more than 110.592 entries, as discovered earlier in the evaluation.

The test was performed in the following logic: within each run, each batch size is executed. Which means *n* routes are inserted and then afterwards deleted. For example, at a batch size of 250, at first 250 insert operations are performed while the duration is measured per operation and afterwards the 250 entries are removed again, while also measuring the delay for each operation. To not measure any setup time, the set of routes that must be inserted and deleted is precomputed within memory. We used /32 prefixes, as they can be simply counted up. Finally, the benchmark was executed for 100 runs at each data plane. While performing the benchmark, the controller was executed on the same host as the forwarder, just as in the test setups before. The cache which is the Tofino switch is connected via a 1 Gbit/s link over the management network of the lab. The results are presented and discussed in the next section per data plane. First for the forwarder and then for the cache.

6.5.1 P4HC DPDK-Forwarder

At first the result of the forwarder is examined. In Figure 6.16 the updates per second as operations per second are shown. In (a) the rates for the insert operations are presented and in (b) the rates for the deletions are provided. As the benchmark only measures the time an operation takes, the rate was calculated by aggregating the duration until one second has elapsed. If a run's complete duration is over 1 second, multiple rate values have been extracted for each run. If a run completes under a second, the rate was scaled up to match one second. The plot also includes the outliers to show that there are not may rates that did not match the average over the 100 runs. However, it can be seen that on smaller batch size more outliers can be observed. Those can be explained due to the computation error on the estimation, if the batch completes in under one second. This can be derived by the observation at a batch size of 5.000 in the plot, where the rate becomes stable without many outliers or jitter. Also, at the batch size of 5.000, an update rate of around 4.600 operations is measured, which indicates that the run has taken longer than one second to complete which will eliminate the computational alignment of the rate. If the rate is estimated before, only a few measurements are present and one longer update could influence the computational estimation. This is resulting in the higher number of outliers and jitter. However, in (a) and (b) can be observed that the average rate has also stabilized between 5.000 and 10.000 for inserts



(a) Insertion rates.

(b) Deletion rates.

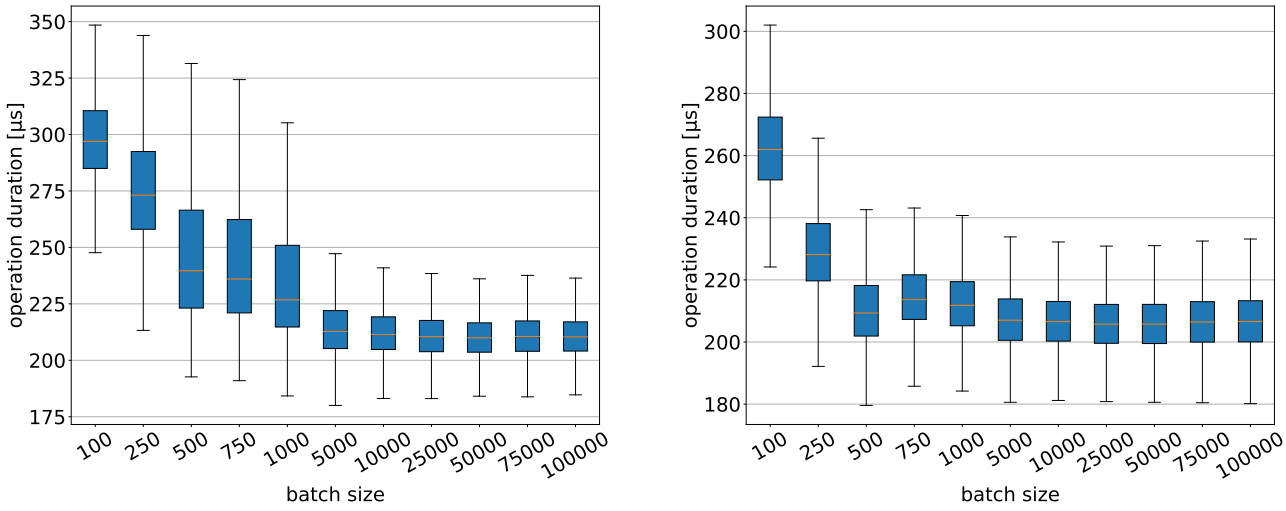
Figure 6.16: Update rates of the DPDK forwarders LPM table.

and deletions. For updates at higher volume, a rate of around 4.700 updates per second can be achieved and for deletion a rate of around 4.800 updates per second. At larger batch sizes the update rate and the deletion rate are coming closer together. Another interesting finding is that the deletion process achieves higher rates in lower batch sizes than the insertion case. This could be explained due to the fact that the delete message itself is smaller than the insert message. The insert message contains a prefix, the egress port and the destination MAC address, while the deletion message only carries the prefix. The stabilization process in (a) could also be explained by the internals of the computing unit. As on smaller updates the compared overhead of fetching data to the memory or the CPU caches requires time. If only a few updates are sent, there are not many updates that can benefit from this. As lager updates happen, the same code gets executed multiple times, which allows the processor for more efficient execution. This effect could also explain the smaller stabilization time in the deletion rate measurements in (b), as the deletion of the prefixes is performed directly after the insertions. So prefetched data in memory or in the CPU caches is beneficial.

Figure 6.17 also provides insights over the duration of an individual operation. The graph (a) and (b) plots the duration of every single operation that was performed during all 100 runs per batch. Again in (a) the insert operations can be seen and in (b) the deletion operations. Within both plots the outliers have been removed, since on larger batches there are a few longer outliers. Those can occur due to scheduling within the operating system, since the communication in gRPC is performed over TCP/IP. However, in general at the insertion case in (a) it can be observed that the inserts in smaller batch sizes are generally taking longer. This could be explained by inefficient processing, compared to larger batches. The duration of an insert also stabilizes at 5.000 updates per batch in terms of the latency, as before on the update rate. In the deletion case on the other hand, it can be observed that the rate stabilizes quicker. Which again indicates that the deletion may be faster or benefit from the prefetching of data, which allows a more efficient processing.

6.5.2 P4HC Cache

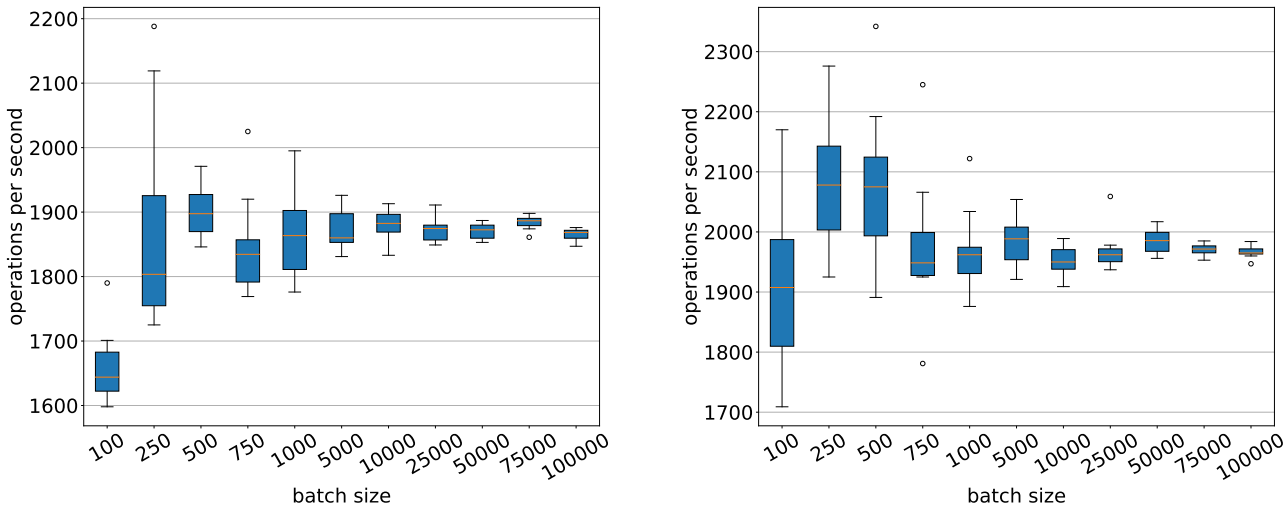
In the last section the result of the forwarder was presented. In this section the results of the cache are presented. All graphs are computed the same way as before in the forwarder section. At the first Figure 6.18 the update rate can be examined. In (a) for the insertions and in (b) for the deletion case. At the first look, it can be seen that in both cases the rate is relatively low compared to the rates at



(a) Insertion duration.

(b) Deletion duration.

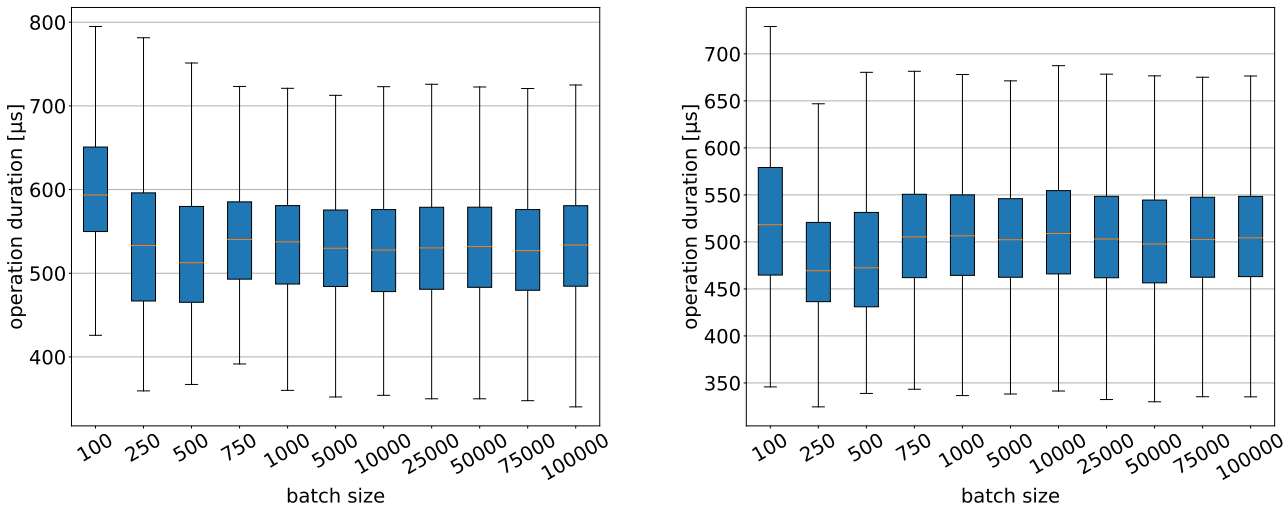
Figure 6.17: Update duration of the DPDK forwarders LPM table.



(a) Insertion rates.

(b) Deletion rates.

Figure 6.18: Update rates of the caches LPM table.



(a) Insertion duration.

(b) Deletion duration.

Figure 6.19: Update duration of the caches LPM table.

the forwarder. On average the forwarder performs more than twice as fast as the cache. However, in comparison the update rates are more constant than those at the forwarder. Beside of the first run, the rates are converging faster to the average rate, compared to the forwarder. Another interesting find is in the deletion case: in (b) it can be observed that smaller delete batches, besides the batch size of 100, perform better than larger batch sizes. This could be explained by the TCAMs internal housekeeping, which becomes more expensive at higher utilization in terms of space. Interesting is that in general the deletion is again faster than the insertion, which could also be an indicator that the gRPC message size influences the performance. However, at larger batch sizes the rates of inserts and deletes are coming closely together with an insert rate of around 1.850 and a delete rate of around 1.950 operations per second. The last interesting finding is that the first batch size of 100 performs significantly worse in comparison the other batch sizes. Here the question again arises, if the setup cost of all involved components is so high that at smaller batches there is more overhead at the first few operations, affecting the rate. As the batch sizes that take fewer than a second are computational normalized, it could be the case that the first few slower measurements have a larger effect than expected.

In Figure 6.19 again all measured durations are plotted. Again in (a) the insert durations and in (b) the deleted ones are plotted. The outliers have also been removed. At the first look in both graphs, the first batch size of 100 performs the worst, as seen before. Afterwards the duration for one operation is almost equal. Also, it can be observed that the deletion in (b) is again faster than the insertion case. As this trend is found in both data planes, there should be only two reasons for this behavior: first the smaller update message size in gRPC and second in general, in TCAM and software LPM tables the deletion of entries is less resource-intensive than inserts. Also interesting is that in the deletion case in (b) the duration, besides the batch size of 100, is lower in small batches. As the duration of one operation takes less time, a higher update rate in the figure before could be seen. Also, in general another reason that the cache performs worse than the forwarder could be the latency that the network is introducing when communicating over the management 1 Gbit/s network to the Tofino switch, as the difference in duration is around 280 nanoseconds on average.

However, in general the question now is what has the most impact on the performance of the update rate. It can be assumed that the effects of different performance at smaller batch sizes compared to larger ones are side effects of the memory technology or housekeeping of the data planes. The question that should be researched in future is what impact does the gRPC protocol has on overall performance. The indicator for future research in this field are first the faster deletion rates, which could be caused by

smaller message sizes. Second that the longer duration at the cache could be introduced by the network communication. However, in this evaluation it was decided to measure the rate from the implemented controller's view. In order to optimize the performance further, an evaluation of the data planes directly and an evaluation of the gRPC protocol overhead could be interesting to figure out where additional performance can be gained. Also it must be noted that every update is sent individually, so it could also be investigated if it is possible to gain more performance when multiple updates are sent within one message.

6.6 Functional P4HC Evaluation

Before the evaluation is summed up, a short evaluation of the general implementation is performed in order to determine if the implantation has met the goals and the design of P4HC. The main goal was to design and implement a protocol independent hybrid switch. The general design idea of P4HC should be able to work with any protocols. Also, the decision to implement the data plane in the P4 language maintains the protocol independency. As the tools are not ready to compile P4 programs to DPDK yet, the forwarder was implemented directly in the DPDK framework. However, since every protocol could be implemented in DPDK and it would be the target of a P4 program, the forwarder is also protocol independent in general. Later on, the forwarder could be replaced, once a functional compiler is available. As the complexity of this work must be reduced, the design and prototype was only developed for IPv4 layer 3 routing, a very common use case. However, the general design, if adjusted, should be compatible with any protocol or offloading use-case. The controller itself is also built up completely modular to support different data planes in the future. Also, the main components are well separated, so those could be replaced or extended easily. The implemented prototype completely implements the specified design. To sum it up, P4HC is the perfect basis to enable further research in hybrid switches.

6.7 Summary

In this chapter the evaluation of the P4HCs data plane has been performed. The evaluation includes all important cases that can happen during a hybrid switch operation. At first it was evaluated how much routes the cache can handle. With the help of the resource allocation output of the compiler, it was possible to free up space for 110.592 routes in the Tofino based cache. This resembles a quite sizeable cache capacity compared to a full table, which has around 850.000 routes. So it would be possible to store around 13 percent of the full table's routes into the cache. Based on the investigation presented in the related work, it has shown that at 100.000 entries it is possible to carry more than 90 percent of a networks traffic. In two traces they have investigated that it would be even possible to carry 99 percent of the traffic. Afterwards the forwarding performance of the cache was investigated. It has shown that the cache enables line rate performance. However, since our test equipment was not fully able to generate line rate data streams at small packet sizes, it can not be confirmed at 100 percent line rate utilization, but it can be assumed based on the remaining measurements. Also, it was evaluated that the average latency that P4HC introduces to a packet at cached routes is around 660 nanoseconds.

Afterwards the forwarder was investigated. It has been worked out that the DPDK forwarder was able to reach 40 Gbit/s line rate at larger packet sizes. Regarding smaller packets, the performance was rather low, since the forwarder implementation uses only one CPU core that maxed out at 5.9 Mpps at 100-byte packet size. This is only around 6 Gbit/s. Also, a quick way to increase the performance was presented and we were able to reach 21,9 Mpps in the test environment. However, the load generator was not able to max out the prototype in this case, since it could only generate three stable flows. The presented performance improvement, based on utilizing multiple CPU cores, scales out only if the packets can be distributed perfectly among CPU cores, which was not possible at only three flows, because the NIC maps

the packets of one flow mostly to one CPU core. Also, during the forwarder's evaluation, the latency of a packet which passes through the forwarder was measured. Since the forwarder could be overloaded, two latency values were provided. If the forwarder is not overloaded, a packet experiences around 8 microseconds of additional latency while traversing trough P4HC. If the forwarder is overloaded, packet loss can occur and the latency addition to non-dropped packets can increase up to around 550 to 700 microseconds on average.

Afterwards the transition when a route is offloaded to the cache was investigated. On the investigation of the transition, the following findings have been made: while offloading a route, out of order packets can be generated due to packets that are still in the forwarders path having a higher latency, compared to the cache path. On an offloading event no packet is getting lost, if no components are overloaded. If only the forwarder is overloaded, there is packet loss to be expected until the cache takes over. In a normal operation, the transition from the forwarder to the cache is instant and the flows can also profit from a much lower latency. An interesting case was found, if a load was applied which almost saturates the 100 Gbit/s port of the cache and the forwarder with its 40 Gbit/s was completely overloaded. After the offloading, an unusual behavior could be observed, which could eventually be explained by filled up buffers, due to an overload of the egress port at P4HC. Now the remaining packets from the cache at 40 Gbit/s and the 100 Gbit/s compete at the 100 Gbit/s egress port. In future work the investigation of this interesting phenomenon should be considered.

Finally, it was investigated how much routes can be inserted to the cache and the forwarder per second. It has been found out that at the cache there are around 1.850 routes that could be added per second and 1.950 routes can be deleted per second. At the forwarder it was possible to insert 4.700 routes per second and remove 4.800 routes per second. While investigating the resulted data, it has cleared out that future work should research for the real bottleneck in the update rate and what influence gRPC has on the update rate. It could also be evaluated, if batch processing of the updates could increase the performance.

However all in all, the results on the performance P4HC has achieved with little to no deeper optimizations is quite impressive, which was not expected in the first place.

7 Conclusions

Hybrid switches are the idea to build one logical switch which consist out of two switches. Mostly one hardware switch is spliced together with a software switch, to overcome the hardware switches limitation like insufficient table space for certain use cases. Based on this idea, this work was started with the questions: Which approaches already exist for hybrid switches and how do they work? and What are the performance characteristics of hybrid switches? Based on those two lead questions, the related work was reviewed. The result was that there are already approaches for hybrid switches. However, as it pointed out, the most works have specialized at the control plane side of the hybrid switches. This is due to the reasons that in hybrid switches complex dependencies between the forwarding rules can arise, which have the possibility to lead to incorrect forwarding behavior, if not handled properly. All approaches did not focus on the data plane side of hybrid switches. Even more, most of the prototypes were only evaluated within a lab and have only used OpenFlow enabled switches in hardware or software. Based on this lack of work at the data plane side, it was decided to focus this work on building a new hybrid switch data plane with state-of-the-art technology to allow future research at the data plane side of hybrid switches. Also, the goal was to not use OpenFlow, as it has many inconveniences like limited extensibility and most switches that support OpenFlow will not allow to implement new protocols, as the switch's hardware cannot be reconfigured after manufacturing.

In this work a new fully programmable data plane approach for hybrid switches was introduced which is called P4HC – P4 Hybrid Cache. P4HC will introduce how a true programmable and protocol independent hybrid switch data plane can be designed and implemented using the P4 language.

7.1 Contribution & Results

As mentioned in the previous section, the main contribution of the work is P4HC as a platform for future research in hybrid switch data planes. P4HC is the first approach of using state of the art technology like P4 and DPDK to create a high performance programmable data plane for hybrid devices. P4HC introduces the concept of two separated data planes in a hybrid switch. One is called the cache and the second one is called the forwarder. The cache is most likely to be implemented on a hardware switch and the forwarder as a software switch on a commodity x86 server. However, since P4HC contains both sides as a dedicated data plane, every other technology could also be utilized to create a hybrid switch.

To create both data planes in a truly protocol independent fashion, the P4 language was used. So the concept describes to create a P4 program for each data plane. This allows P4HC to be fully platform independent, since P4 could be compiled to many target platforms while enabling full programmability to implement any network protocol. As a cache data plane a P4-programmable Tofino switch was used. At the forwarder side, the approach of compiling an P4 program to DPDK was investigated. DPDK is a software framework that allows line rate packet forwarding on commodity x86 hardware. Unfortunately, it was discovered that the existing approaches of compiling P4 to DPDK are not ready to use at the time of writing. So, it was required to implement the forwarder directly in DPDK. Also, the unknown egress port problem was introduced, which arises in hybrid data planes. As the cache locates all connections to the network, the forwarder must be able to signal the correct egress port to the cache, if it has made the forwarding decision. Otherwise, the packet would loop forever, if the cache has no matching rule. As the focus was at the data plane, it was decided to implement P4HC to cover the use-case of layer 3 IPv4 routing. This allows to reduce the complexity of P4HC, which in turn allows to optimize the data plane further and simplifies the controller. Also, it covers one of the most common use cases on the Internet: full table routing. A controller was implemented to interface with both data planes. The connection to the data planes was realized through gRPC. Also, it was paid attention to implement the controller in a modular way, so it is possible to implement other data planes later for future research. To exchange

route information with neighbor routers, a BGP daemon was integrated and a simple dependency resolve algorithm was created to resolve dependencies between individual routes. Finally, for the first time it was possible to evaluate a modern high performance data plane of a hybrid switch that wants to compete against enterprise grade routers.

During the evaluation of P4HC, it could be proven that it is possible to create a line-rate capable data plane for hybrid switches with plenty of table-space to store an Internet full table. The used Tofino switch was able to store 110.592 IPv4 routes and the forwarder can store as many routes as the memory can hold. In our tests we have allocated a fixed table for 1 million routes, in order to store a full table. During the evaluation, the concept of P4HC was proven to be functional. The cache was able to route packets at 100 Gbit/s line rate. The forwarder has also shown the performance potential of software forwarding. It was possible to serve 40 Gbit/s at line-rate at large packet sizes. However, as it points out smaller packets at higher rates are much more complicated to forward at line-rate. At the first attempt it was possible to forward 5.9 Mpps at one CPU core. To scale up the performance, the utilization of multiple CPU cores has been implemented. Unfortunately, we were not able to serve 40 Gbit/s line rate at 100-byte packets. As it points out, the forwarding performance does not scale based only on the number of CPU cores. It was found out that the performance gain at multiple CPU cores also depends on the ability to distribute the packets to the multiple cores evenly. As the distribution is based on a per flow-based hash, the forwarder's performance will only scale out on more parallel flows. However, the used load generator was not able to create stable measurements at more than three flows and thus creates a performance limit at 21.8 Mpps. So, one important finding at this point is that a hybrid architecture with a software data plane performs better at multiple flows, compared to a single elephant flow. As another performance attribute, the introduced latency of a hybrid switch was evaluated. If a route is installed into the cache, a packet experiences a latency of around 660 nanoseconds. When it traverses the forwarder, the latency increases to around 8 microseconds, if the forwarder is not overloaded. Finally, also a benchmark was built and contributed to measure the update rates of the data planes' forwarding rules. The benchmark is implemented in a generic way, which allows to reuse the benchmark for future research on other data planes that could be integrated into P4HC later. Finally, our evaluation has investigated for the first time the transition behavior of a hybrid switch. It was found out that, as long as the forwarder will not be overloaded, a route can be installed into the cache without any major problem. The only problem that could be found are out of order packets, which should not be an issue, since the most networks will in general not guarantee in order delivery. The case where the cache is overloaded due to routes being offloaded, it shows interesting behavior that could be challenged at future research.

To sum it up P4HC proves the concept that it is possible to build a high-performance data plane for hybrid switches. This allows to replace specialized networking hardware with a hybrid device, if the following conditions are met: First the network has no problem with the additional latency introduced by the forwarder. Second the network does not only have single high pps flows. And finally, the network does not require more forwarding changes per second than both data planes can handle. Also, P4HC contributes a platform for performing future research on programmable hybrid switches to the community.

7.2 Future Work

Within this work we have found interesting behaviors that should be faced in future research. The first issue that should be considered is the research on P4 language to DPDK compilers. This would allow to implement a complete data plane in P4 for hybrid switches. The benefit would be to have a complete data plane in one common language and being able to perform easy adjustments, without deep diving into DPDK code. Also, we faced the issue of lacking forwarding performance on single high pps flows. It should be researched how this limitation could be overcome. During investigation of the transition zone when an offload happens, an unexpected behavior was found. The hypothesis is that this only happens if the cache gets overloaded when performing the offloading event. The behavior of this case

should be further investigated and future research could be performed to find a solution to mitigate the problem. Also, while investigating the update rates of the data planes, there are indicators that maybe gRPC could be influencing the performance negatively. It should be researched how much performance is lost through the usage of gRPC. Furthermore, it could be researched if the performance could be improved when bulk updates are used instead of single updates. Also, it would be interesting to evaluate the update rate of the data plane in real word networks, to check if a hybrid data plane could handle the resulting updates to both data planes, even when the dependency resolving process adds additional updates. Another idea would be to research how the software data plane part performs when many rules are installed and many flows will require most of the installed rules. Is the performance suffering under this condition, because the routes could be not cached efficiently by the CPU and continuous lookups for the rules in the system memory would be the consequence? There are also many improvements that could be performed at P4HC, like implementing a more efficient dependency resolving strategy, implement an ARP resolver at the controller that will learn the next hop MAC addresses based on ARP, instead of reading out the configuration file. It could also be implemented that BGP is directly handled through the cache's port, so that the BGP traffic is forwarded from the data plane to the controller. Finally, it would be interesting to research other technologies that could be used for one of the data planes. Like for example a SmartNIC as a cache, if not too many ports at the cache are required.

Bibliography

[AB12] R. Atkinson and SN Bhatti. Address Resolution Protocol (ARP) for the Identifier-Locator Network Protocol for IPv4 (ILNPv4). RFC 6747, 2012.

[ACS03] Igor Arsovski, Trevis Chandler, and Ali Sheikholeslami. A ternary content-addressable memory (tcam) based on 4t static storage and including a current-race sensing scheme. *IEEE Journal of Solid-State Circuits*, 38(1):155–158, 2003.

[ARS16] M. Agiwal, A. Roy, and N. Saxena. Next generation 5g wireless networks: A comprehensive survey. *IEEE Communications Surveys Tutorials*, 18(3):1617–1655, 2016.

[BBCC14] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. Openstate: Programming platform-independent stateful openflow applications inside the switch. *SIGCOMM Comput. Commun. Rev.*, 44(2):44–51, April 2014.

[BBRK16] A. Blenk, A. Basta, M. Reisslein, and W. Kellerer. Survey on Network Virtualization Hypervisors for Software Defined Networking. *IEEE Communications Surveys Tutorials*, 18(1), 2016.

[BD17] Mihai Budiu and Chris Dodd. The p416 programming language. *SIGOPS Oper. Syst. Rev.*, 51(1):5–14, September 2017.

[BDG+14] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[bfna] Barefoot networks - the worldś fastest & most programmable networks. `https://barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/`. Accessed: 2020-10-15.

[bfnb] Barefoot networks - tofino. `https://barefootnetworks.com/products/brief-tofino/`. Accessed: 2020-10-15.

[Bin06] Benny Bing. Ubiquitous broadband access networks with peer-to-peer application support. *Evolving the Access Network*, pages 27–36, 2006.

[bio] bio-rd: A re-implementation of bgp, is-is and ospf in go. `https://github.com/bio-routing/bio-rd`. Accessed: 2020-11-20.

[BM15] Roberto Bifulco and Anton Matsiuk. Towards scalable sdn switches: Enabling faster flow table entries installation. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 343–344. Association for Computing Machinery, 2015.

[BM19] Roberto Bifulco and Anton Matsiuk. Software-enhanced stateful switching architecture, January 10 2019. US Patent App. 16/068,115.

[bmv] Behavioral model (bmv2). `https://github.com/p4lang/behavioral-model`. Accessed: 2020-10-15.

[Bra17] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, December 2017.

[cid] Cidr Report. https://www.cidr-report.org/as2.0/. Accessed: 2020-10-11.

[CSF13] Nikolaos Chatzis, Georgios Smaragdakis, and Anja Feldmann. On the importance of internet exchange points for today's internet ecosystem. *CoRR*, abs/1307.5264, 2013.

[DH17] Dr. Steve E. Deering and Bob Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 8200, July 2017.

[dpda] Dpdk 20.05 documentation - rss. https://doc.dpdk.org/guides-20.05/nics/features.html?#rss-hash. Accessed: 2020-12-14.

[dpdb] Dpdk Data plane development kit. https://www.dpdk.org/. Accessed: 2020-10-11.

[ETS13] GSNFV ETSI. Network functions virtualisation (nfv): Architectural framework. *ETsI Gs NFV*, 2(2):V1, 2013.

[FHF+11] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. *ACM Sigplan Notices*, 46(9):279–291, 2011.

[FLYV93] Vince Fuller, Tony Li, Jessica Yu, and Kannan Varadhan. Rfc1519: Classless inter-domain routing (cidr): an address assignment and aggregation strategy, 1993.

[FPEM17] Xenofon Foukas, Georgios Patounas, Ahmed Elmokashfi, and Mahesh K Marina. Network slicing in 5g: Survey and challenges. *IEEE Communications Magazine*, 55(5):94–100, 2017.

[go] Go. https://golang.org/. Accessed: 2020-11-20.

[grp] grpc - a high-performance, open source universal rpc framework. https://grpc.io/. Accessed: 2020-11-20.

[HB96] John Hawkinson and Tony Bates. Rfc1930: Guidelines for creation, selection, and registration of an autonomous system (as), 1996.

[Hec07] Oliver M Heckmann. *The competitive Internet service provider: network architecture, interconnection, traffic engineering and network design*. John Wiley & Sons, 2007.

[i26] Intel xeon prozessor e5-2650 v4. https://ark.intel.com/content/www/de/de/ark/products/91767/intel-xeon-processor-e5-2650-v4-30m-cache-2-20-ghz.html. Accessed: 2020-10-12.

[infa] Influxdb: Purpose-built open source time series database. https://www.influxdata.com/. Accessed: 2020-11-20.

[infb] Influxdb-python. https://github.com/influxdata/influxdb-python. Accessed: 2020-11-20.

[infc] influxdb1-clientv2. https://github.com/influxdata/influxdb1-client. Accessed: 2020-11-20.

[infd] Influxql: Influx query language. https://docs.influxdata.com/influxdb/v1.8/query_language/. Accessed: 2020-11-20.

[JGRW15] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. Covisor: A compositional hypervisor for software-defined networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 87–101. USENIX Association, 2015.

[JLZ+17] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 121–136. Association for Computing Machinery, 2017.

[JLZ+18] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49. USENIX Association, 2018.

[KARW16] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proceedings of the Symposium on SDN Research*, pages 1–12. Association for Computing Machinery, 2016.

[KNB+20] Ralf Kundel, Leonhard Nobach, Jeremias Blendin, Wilfried Maas, Andreas Zimber, Hans-Joerg Kolbe, Georg Schyguda, Vladimir Gurevich, Rhaban Hark, Boris Koldehofe, and Ralf Steinmetz. Openbng: Central office network functions on programmable data plane hardware. *International Journal of Network Management*, page e2134, 2020.

[KRV+15] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-Defined Networking: A Comprehensive Survey. *Proceedings of the IEEE*, 103(1), 2015.

[KSB+20] R. Kundel, F. Siegmund, J. Blendin, A. Rizk, and B. Koldehofe. P4sta: High performance packet timestamping with programmable packet processors. In *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9, 2020.

[KUAh+20] Ibrahim Kettaneh, Sreeharsha Udayashankar, Ashraf Abdel-hadi, Robin Grosman, and Samer Al-Kiswany. Falcon: Low latency, network-accelerated scheduling. EuroP4'20, pages 7–12, New York, NY, USA, 2020. Association for Computing Machinery.

[l2f] Dpdk example application l2fwd. `https://github.com/DPDK/dpdk/tree/main/examples/l2fwd`. Accessed: 2020-8-1.

[LAW13] Yaoqing Liu, Syed Obaid Amin, and Lan Wang. Efficient fib caching using minimal non-overlapping prefixes. *SIGCOMM Comput. Commun. Rev.*, 43(1):14–21, 2013.

[LAW15] Yaoqing Liu, Syed Amin, and Lan Wang. Efficient fib caching using minimal non-overlapping prefixes. In *Computer Networks*, volume 83, pages 85–99. Elsevier, 2015.

[LC15] Y. Li and M. Chen. Software-defined network function virtualization: A survey. *IEEE Access*, 3:2542–2553, 2015.

[LCS+19] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 318–333. Association for Computing Machinery, 2019.

[lib] Libpcap. `https://www.tcpdump.org/`. Accessed: 2020-11-20.

[log] Logrus. `https://github.com/sirupsen/logrus`. Accessed: 2020-11-20.

[MAB+08] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2), 2008.

[mel] Mellanox NICs Performance Report with DPDK 19.08. Technical report, Mellanox Technologies. Accessed: 2020-06-20.

[min] Mininet - an instant virtual network on your laptop (or other pc). http://mininet.org/. Accessed: 2020-10-15.

[MJ93] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference*, USENIX'93, page 2. USENIX Association, 1993.

[MSG+16] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. De Turck, and R. Boutaba. Network function virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys Tutorials*, 18(1):236–262, 2016.

[mxj] Juniper: Universelle routing-plattform der mx-serie. https://www.juniper.net/de/de/products-services/routing/mx-series/. Accessed: 2020-12-11.

[net] Netronome nfp-4000 flow processor. https://www.netronome.com/m/documents/PB_NFP-4000-7-20.pdf. Accessed: 2020-06-20.

[nex] TheNextPlatform: The walls come down on the last bastion of proprietary. https://www.nextplatform.com/2016/06/14/walls-come-last-bastion-proprietary/. Accessed: 2020-12-11.

[nra] nradix: Translated to golang implementation of nginx's radix tree. https://github.com/asergeyev/nradix. Accessed: 2020-11-20.

[OTCK20] T. Osiski, H. Tarasiuk, P. Chaignon, and M. Kossakowski. P4rt-ovs: Programming protocol-independent, runtime extensions for open vswitch with p4. In *2020 IFIP Networking Conference (Networking)*, pages 413–421. IEEE, 2020.

[Ozd12] R. Ozdag. Ethernet Switch FM6000 Series - Software Defined Networking. Technical report, Intel Corporation, 2012.

[p41a] P4_14 language specification. https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf. Accessed: 2020-06-20.

[p41b] P4_16 language specification. https://p4.org/p4-spec/docs/P4-16-v1.2.1.pdf. Accessed: 2020-10-15.

[p41c] P4_16 language tutorial. https://p4.org/assets/p4_d2_2017_p4_16_tutorial.pdf. Accessed: 2020-06-20.

[p4c] p4c-ubpf: a new back-end for the p4 compiler. https://p4.org/p4/p4c-ubpf. Accessed: 2020-10-15.

[p4o] P4 language consortium. https://p4.org/. Accessed: 2020-06-20.

[p4ra] P4rt-ovs: Programming protocol-independent, runtime extensions for open vswitch using p4. https://github.com/0range-0penSource/p4rt-ovs. Accessed: 2020-10-15.

[p4rb] P4runtime specification. https://github.com/p4lang/p4runtime. Accessed: 2020-11-20.

[pena] Improving DPDK Performance. Technical report, NETCOPE TECHNOLOGIES. Accessed: 2020-06-20.

[penb] Pensando dsc-100 distributed services card. https://pensando.io/wp-content/uploads/2020/03/Pensando-DSC-100-Product-Brief.pdf. Accessed: 2020-12-11.

[pma] pmacct. http://www.pmacct.net/. Accessed: 2020-11-20.

[PMP01] Sonia Panchen, Neil McKee, and Peter Phaal. InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. RFC 3176, September 2001.

[PPK+15] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. The design and implementation of open vswitch. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 117–130. USENIX Association, 2015.

[pyt] python. https://www.python.org/. Accessed: 2020-11-20.

[rfc81] Internet Protocol. RFC 791, September 1981.

[RHL06] Yakov Rekhter, Susan Hares, and Tony Li. A Border Gateway Protocol 4 (BGP-4). RFC 4271, January 2006.

[RLH+94] Yakov Rekhter, Tony Li, Susan Hares, et al. Ffc4271: A border gateway protocol 4 (bgp-4), 1994.

[RMF+13] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. Modular sdn programming with pyretic. *Technical Reprot of USENIX*, 2013.

[ryu] Ryu. https://osrg.github.io/ryu/. Accessed: 2020-06-20.

[SFU+10] Nadi Sarrar, Anja Feldmann, Steve Uhlig, Rob Sherwood, and Xin Huang. Towards hardware accelerated software routers. In *Proceedings of the ACM CoNEXT Student Workshop*, CoNEXT '10 Student Workshop. Association for Computing Machinery, 2010.

[SUF+12] Nadi Sarrar, Steve Uhlig, Anja Feldmann, Rob Sherwood, and Xin Huang. Leveraging zipf's law for traffic offloading. *Computer Communication Review*, 42(1):17–22, 2012.

[ubp] Userspace ebpf vm. https://github.com/iovisor/ubpf. Accessed: 2020-10-15.

[VHK+18] P. Vörös, D. Horpácsi, R. Kitlei, D. Leskó, M. Tejfel, and S. Laki. T4p4s: A target-independent compiler for protocol-independent packet processors. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–8, 2018.

[WDF+05] Jörg Wallerich, Holger Dreger, Anja Feldmann, Balachander Krishnamurthy, and Walter Willinger. A methodology for studying persistency aspects of internet flows. *SIGCOMM Comput. Commun. Rev.*, 35(2):23–36, April 2005.

[wed] Edge-core networks: 100gbe data center switch bare-metal hardware wedge100bf-32x/65x. https://www.edge-core.com/_upload/images/Wedge100BF-32X_65X_DS_R07_20200731.pdf. Accessed: 2020-10-12.

[x70] 7060x and 7260x series 10/25/40/50/100g data center switches. https://www.arista.com/assets/data/pdf/Datasheets/7060X_7260X_DS.pdf. Accessed: 2020-10-12.

[xl7] Intel Ethernet-Converged-Network-Aapter XL710-QDA1. https://www.intel.de/content/www/de/de/products/docs/network-io/ethernet/network-adapters/ethernet-xl710-brief.html. Accessed: 2020-10-12.

[ZCQZ04] Tanja Zseby, Benoit Claise, Juergen Quittek, and Sebastian Zander. Requirements for IP Flow Information Export (IPFIX). RFC 3917, October 2004.

[Zim80] H. Zimmermann. Osi reference model - the iso model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, 1980.

[ZLZ+16] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. OpenNetVM: A platform for high performance network service chains. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMIddlebox '16, pages 26–31. Association for Computing Machinery, 2016.