CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Integrating Programmable Smart-NICs into Industrial Packet-Processing Systems

Master's thesis in Computer science and engineering

Lina Blomkvist and Tove Svensson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Integrating Programmable Smart-NICs into Industrial Packet-Processing Systems

Lina Blomkvist and Tove Svensson

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Integrating Programmable Smart-NICs into Industrial Packet-Processing Systems

Lina Blomkvist and Tove Svensson

Supervisor: Romaric Duvignau, CSE
Advisor: Patrik Nyman and Eric Nordström, Ericsson
Examiner: Marina Papatriantafilou, CSE

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2021

Integrating Programmable Smart-NICs into Industrial Packet-Processing Systems

Lina Blomkvist and Tove Svensson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

In order to cope with the requirements of 5G, Smart Network Interface Controllers are being used to offload general use CPUs. For them to be able to become more widely used, research of how to integrate them into already existing industrial systems and the cost of such a transition is needed. This thesis presents the method of integrating a P4 programmed Netronome Agilio Smart Network Interface Controller (SNIC) into a high speed industrial packet processing pipeline. A partition of the industrial system handling packet classification is translated and implemented and run on the SNIC and tested and compared to the original program performing the same task. The effect on performance is analysed and a qualitative evaluation of the process conducted. The challenges faced in this project consisted of understanding the industrial system environment and how to seamlessly translate the code without losing the original functionality.

Keywords: 5G, P4, SDN, SNIC.

Acknowledgements

Thank you to Romaric Duvignau, Patrik Nyman, and Erik Nordström for making this thesis possible, and to Dhrupal Hareshkumar Tilava, for your incredible support during the project.

Lina Blomkvist Tove Svensson, Gothenburg, November 2021

Contents

List of Figures

1

Introduction

With the growing network transmission speeds of 5G, packet handling using general-use CPUs is a bottle-neck in regard to speed. To improve the performance in packet handling, Network Interface Controllers (NIC) are used to handle full or partial packet offload from the CPUs in data centers. Furthermore, to be able to adapt to clients' changing demands on features, programmable NICs, called Smart-NICs (SNIC), are increasingly being used.

The SNICs are able to be programmed using various programming languages, an example of such is P4 [1], which is utilized by many large companies including Cisco, Intel, and Google [2]. P4 is a domain specific language for network programming and was developed to improve the flexibility and reconfigurability of networks. These capabilities, in combination with the programmable SNIC, are crucial to be able to apply Software Defined Networking (SDN) which includes a control plane allowing change of network functionality without physical interference. The possiblity to use SDNs is key to being able to adapt to new technology and integrate new functionalities efficiently, which is why it is necessary to research the feasibility of replacing existing NICs in exiting data centers, with SNICs.

Though the possible performance gains of using SNICs have been known for a couple of years, little research on the integration process have been conducted. Research is required to investigate the method of integration, limitations to the type of and amount of logic that can be transferred the Smart-NIC, as well as the impact the integration has on performance. During this project, a part of an C-programmed 5G industrial packet processing pipeline was translated to P4 to then be offloaded to a P4-programmed SNIC. The performance the system gained by using the SNIC was then evaluated against the challenges the translation process brought to arrive at a final evaluation. The project strived to further the understanding of the SNIC integration process and the possibilities and limitations that it entails.

1.1 Problem

In order for the use of P4-programmed SNICs to increase, it has to be evident that the performance gain in using SNICs is greater than the cost of transitioning to using them. This stresses the need to investigate the challenges in the process of offloading an industrial pipeline to a SNIC using P4. For a system operating mainly on CPUs or regular NICs to transition to P4-programmed SNICs, limiting factors of the language in terms of functionality must be identified in order to conclude which parts of the pipeline can and cannot be translated. Furthermore, it is necessary to evaluate the performance of the offloaded functions in a real system to ensure that the research suggesting that the use of SNICs with P4 will improve the performance of a system, holds up in practice.

With this in mind, the problem statement of the project has been phrased as the following research questions:

1. Do the performance gains justify the costs when integrating a SNIC to offload the CPU/ASICs of packet processing in an industrial cloud native data plane?
2. What are the limiting factors when using P4 programmed SNICs as opposed to CPU/ASICs?
3. Does the limited scope experiments performed in testing environments resulting in higher performance hold up in real scenarios?

A summary of current research within the area of using Smart-NICs can be found in Section 2.7.

1.2 Goals

In order to answer the research questions stated in the previous section, three goals for the project have been outlined. Presented below, these goals involve translating a part of an industrial packet processing pipeline to P4 and running it on a SNIC. To ensure that the functionality is intact and measure the performance of the resulting system rigorous testing must be performed. The evaluation of the project should account for the performance of the system, and there should be a qualitative evaluation of the difficulty of the process and limitations of the technology.

The goals of the project have been summarised as follows:

1. Identify a part of the cutting edge industrial packet processing pipeline suitable for offloading to a SNIC and translate the logic of this section into a P4 program.
2. Offload part of the initial production level pipeline to a SNIC while preserving the functionality of the system.
3. Evaluate the performance of the resulting system with consideration to the

limits and difficulties of the process.

1.3 Challenges

For the choice of which part of the system logic would be used for translation into a P4 program to control the SNIC, the guidance of industrial experts is required. The specific functionality of this part to be converted is also pinpointed by experts. However, in order to understand how the chosen logic communicated with its environment and how to mimic this in the translation, further investigation has to take place. Since the system is large and complicated, the main challenge in this is to understand the environment within which the chosen logic is deployed.

After the logic had been translated, the next challenge consisted of integrating the use of the SNIC with the P4 code into the industrial system. This challenge included preserving the dynamicity of the previous code and finding a way to communicate between the SNIC and the original C code at an acceptable speed.

1.4 Method

One of the main goals of this project regards evaluating the process of the implementation of a P4 program and integration of a SNIC into a large pipeline. In order to perform the evaluation in an ordered manner, the methodology of the project requires careful consideration. Each step of the process can be isolated to ensure it is clear at what stage any issue arises and what challenges are associated with each part of the process. The details of the development is further explained in Section 3 and 4, but the following list gives an overview of the projects methodology.

1. A development environment was set up where a P4 programmed switch and a few end hosts were simulated using a network simulation software. This environment was used to test the functionality of the program during the implementation, before testing on real hardware.

2. Before translation of the industry code, an initial P4 program was created. The program did not include any specific functionality, but was rather just a shell to make sure parsing and deparsing of packets worked correctly and the development environment was operating as intended.

3. A suitable part of the industry pipeline was then selected for translation, and the initial P4 program was extended to accommodate this. A custom output header was introduced to deliver the results of the computations, and the functionality of the industry code translated to P4.

4. The SNIC was then loaded with the code and put in a testing environment to simulate the actual system. From which the performance was measured and compared to that of the initial system's.

5. After this, a qualitative evaluation was performed, discussing the difficulty of

the integration process.

6. The final result was then concluded by weighing the results of the qualitative evaluation and the performance evaluation against each-other.

1.5 Limitations

There are several limitations in measuring the impact Smart-NIC usage may have on performance in an industrial system. For example, due to lack of time and resources, the research of this project is based on tests performed in one particular system only. Therefore, it is not known if the performance of systems structured differently will be affected by Smart-NIC usage in the same manner. Naturally, the difficulty of Smart-NIC integration in other systems is also unknown.

Another limitation to the performance impact measurements is that the project is limited to using one particular Smart-NIC hardware, again due to time and resource limitations. Hence, it could be that other Smart-NIC devices could affect performance differently.

The evaluation is also limited in scope, in that the main points of interest chosen are performance and ease of implementation. As such, other aspects such as the cost of the respective hardware or the difference in energy consumption of the devices has not been considered.

Further, the project is limited to handling only one part of a larger pipeline. The pipeline handles packet processing in a large scale industrial 5G network, and the part that is examined in this project performs simple classification of incoming packets. The reason for choosing only this part of the pipeline is primarily due to time constraints but also due to limitations of the use of P4 in the Smart-NIC. While P4 offers great flexibility, it is not possible to inspect the application layer of a packet in a P4 program, making in unable to perform operations that rely on this, for this reason, some parts of the industrial pipeline considered in this project are not possible to offload to a P4 programmed SNIC. There is hence a possibility that other programming languages could have other possibilities and thus effects on performance. However, due to time constraints, the choice was made not to investigate any additional languages.

1.6 Report structure

The thesis begins by describing the needed background in Chapter 2. This chapter starts with Section 2.1 which describes the requirements on 5G and its architecture. Thereafter, Section 2.2 explains the workings of NIC to then introduce Smart-NIC by first motivating the need for them followed by an architectural description and an illustration of a packet's path through a SNIC. Section 2.4 then describes the use of the P4 language and its functionality. The background continues by Section 2.6 where additional software used in the project is briefly introduced. The final section

discusses related works to illustrate the current state of research in SNIC and P4 usage.

Chapter 3 regards the implementation of the project, including the setup of the environment, translation of the C code to P4 and, the integration of the program into the industrial pipeline. Following in Chapter 4 the evaluation of the project is depicted. The testing environment is presented along with the testing criteria and methodology, followed by a description of the qualitative evaluation criteria.

In Chapter 5 the results of the tests done are described in detail along with the assessment of the qualitative evaluation criteria. Finally, Chapter 6 contains the conclusions drawn by the authors as well as a discussion of the results.

2

Background

The fifth generation cellular networks (5G) have to satisfy the needs of the coming years. The amount of traffic will not only significantly increase, but also introduce new demands as the nature of the connected devices change and the IoT grows [3]. In order to support the growth in uses such as cloud based technology, virtual streaming (vstreaming), and online gaming as well as the significant increase of Internet of Things (IoT) devices, demands on the networks are very high. The 5G networks need to satisfy low latency as well as a huge increase in bandwidth from 4G, while still pursuing to further decrease in energy consumption [4]. In order to meet the new requirements, the underlying architecture of the network needs to be both efficient and flexible. It is demonstrated in simulations that Smart-NICs could offer a possible solution to this problem [5, 6, 7, 8, 9, 10]. P4 is a new and increasingly popular language for programming these Smart-NICs.

This chapter gives an introduction to 5G networks followed by an explanation of what a SNIC is and how it works. Thereafter, the P4 language and other software tools used in the project are presented. Finally, there is a summary of relevant related work.

2.1 Requirements of 5G

This Section begins by giving a closer introduction to the requirements of 5G, briefly explaining the main solution concepts. The following sections goes deeper into how the architecture of 5G vows to fulfill the requirements.

2.1.1 5G Use-Cases

The 5G network is designed with vertical industries (markets catering to specific industries) in mind. These industries, including industrial automation, mission critical IoT, and medical care, come with new network use cases which put a new kind of pressure on the network architecture [11].

The different services of each industry demand appropriate data- and network resource allocation. Mission critical IoT for example, may contain services needing certain resources guaranteed at all time to be able to pull of critical tasks without being interrupted by network traffic. The solution to this is for the 5G architecture

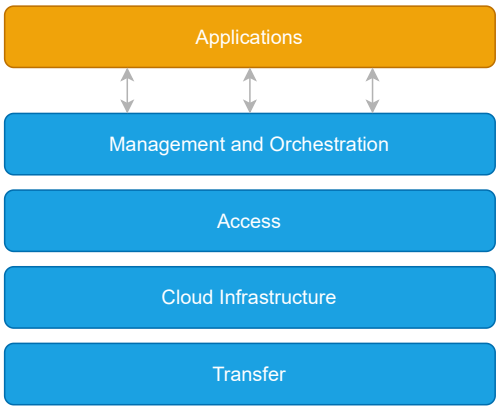

Figure 2.1: The layers of the 5G infrastructure.

to provide separate logical networks per industry. The separate logical networks are called network slices and each separate network slice contains functions specialized in the field it is managing. In order to handle the complex allocation management of resources slices incur, telecommunication operators deploy special orchestration functions. Further, if an operator lacks in a resources, or certain services, there is a need for the possibility of cross-domain orchestration, which also pushes the limits of the 5G architecture [12].

One of the main concepts of 5G is to facilitate deployment, management, and creation of new services. To be able to provide for this a Service-Based Architecture (SBA) is used, providing applications as a set of micro-services. For optimal performance, these micro-services should be further deployed in a so cloud-native 5G core network, where cloud services are used to the utmost ability to decouple networks and network functionality from hardware. For example, router functions could be moved from hardware to a remote virtual machine making it easier to deploy and remove [13].

In order to accommodate for a cloud networking environment, the 5G architecture further needs to integrate a way to manage hardware functionality in software, and an infrastructure where functionality within the hardware may be transferred to the cloud. Together, these infrastructures make cloud-native networks possible. These methods also promote creation of automation software to minimize manual handling of the 5G network [12, 11].

2.1.2 5G Architecture

The general structure of the 5G Architecture is defined as shown in Figure 2.1 where the horizontal boxes represent the different infrastructure layers. A more detailed view of how the different layers are connected can be seen in Figure 2.2, where the user plane, which is the part this project concerns, is encompassed by a blue line.

The top layer is the application layer. This layer contains the applications using the network for their services. An application in the 5G network is defined as a software that uses underlying micro-services for it is provided functionality. This approach,

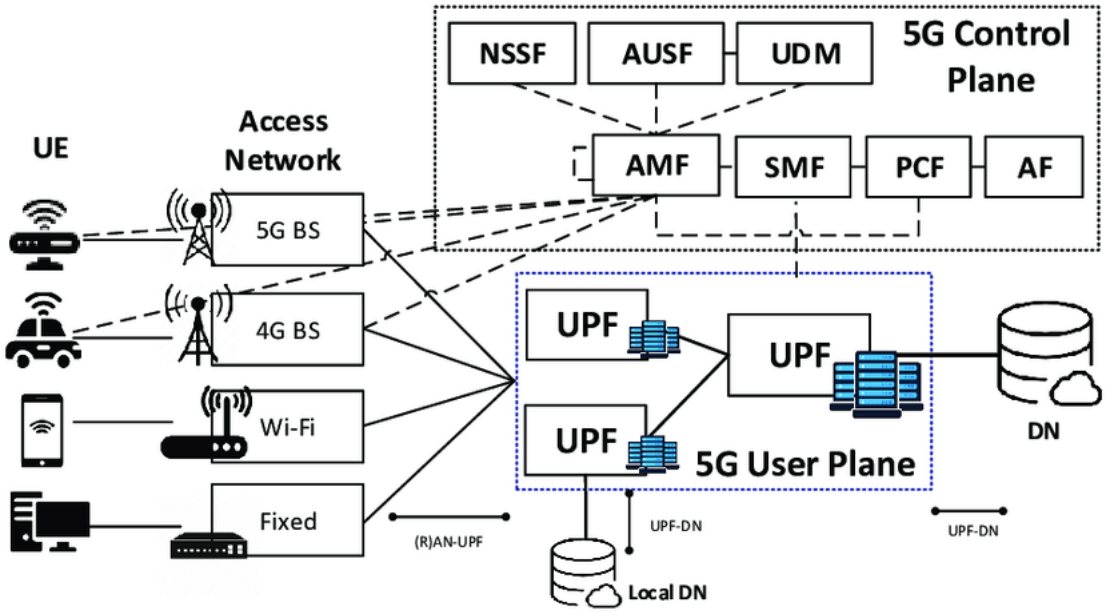

Figure 2.2: More detailed view over the different layers [14].

shown in Figure 2.3, steps away from the previous monolithic approach where each application is constructed as a single unit. The benefits of the modular approach includes the ability to scale only the micro-services in demand when scaling the application and the simplicity in only having to rebuild and deploy a new micro-service instead of the whole unit when making small changes to the code. The micro-service architecture also makes it easier to distribute and maintain services without needing large amounts of processing power and memory [15, 12, 16].

The challenge of using a micro-service architecture is the large amount of orchestration needed to maintain it. This is where the management and orchestration layer comes into play. A part of this layer manages network functions and interfaces, to for example micro-services, by available physical and virtual resource. Another part of the management and orchestration layer concerns maintaining the software defined control plane needed for SDNs. This control plane moves away control from hardware by making it possible to manage devices beyond physical connectivity via control software. By providing open API through software, the SDN control plane enables central programming of network behavior which in turn provides increased simplicity in management for operators. Using SDN also provides network automation which orchestrates the direction of network traffic depending on certain conditions [15, 12, 16].

With SDNs, NFV becomes more compelling. NFV focuses on the decoupling of network functions and hardware devices by using virtual machines on standardized (non-specialized) hardware. This eliminates need for specialized hardware and creates the possibility of using one machine for multiple network functions. Another benefit of NFV is to be able to move network functions around as demand changes, increasing flexibility. For example, if a new network function is needed, a new virtual

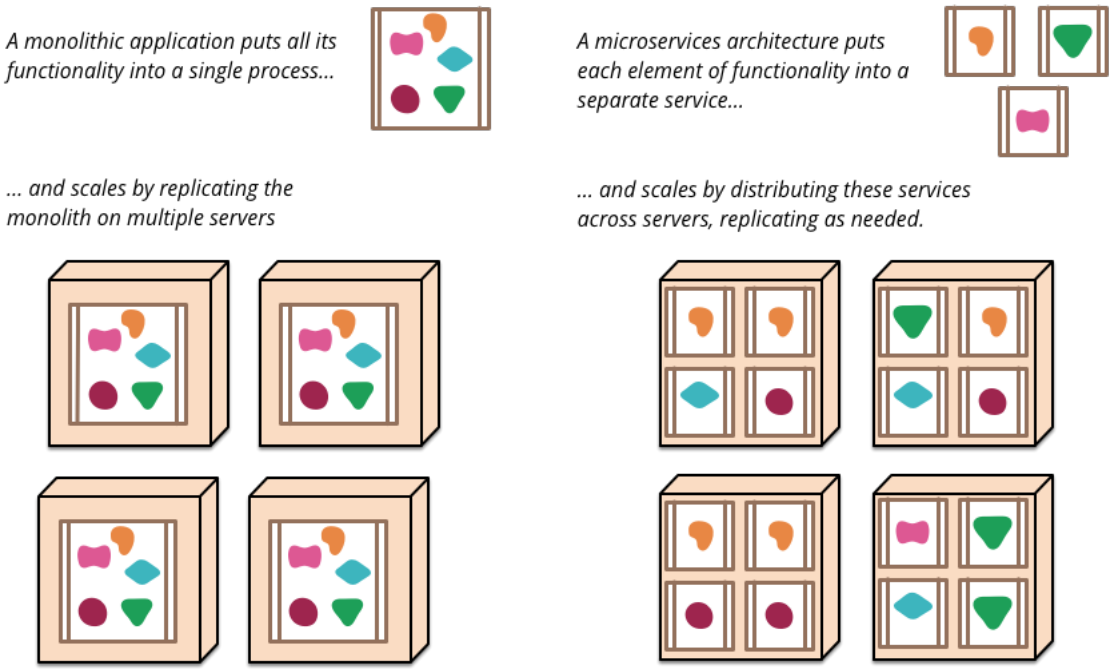

Figure 2.3: Explanation of the micro-service architecture compared to the monolithic approach [15].

machine is simply set up and when it is not needed the Virtual Machine (VM) is removed. The NFV resource and network demands are also orchestrated by the second layer and together with the SDN approach a network decoupled from hardware from functions to control is created [17, 18, 19].

Another concept of the 5G architecture is to enable network community innovation, making it possible for anyone to use the networks capabilities to evolve network functionality. In order to do this, the second layer also includes control functions for network exposure, meaning to present an API for programmer to use the provided services [16, 20].

The access layer contains the access and communication services and the packet core. The access services established access to the edge nodes of the network and distributes data plane functions to the edges of common core networks. The communication services manage how services communicate over different service providers as well as to a client over a single provider. It further defines how IP Multimedia Services (IMS) are distributed in the new cloud environment as well as how inter-networking with 3G and 4G is executed [16, 21].

The third layer contains the packet core. The packet core separates different data channels between an IP and user equipment (UE) in order to set parameters regarding performance. Thereafter, the PDN (Packet Data Network) Gateway is set as an IP anchor point for IP-communication between the UE and external PDNs (the Internet). The PDN Gateway also provides packet filtering and policy enforcement such as Quality of Service (QoS). The Serving Gateway (SGW) is the mobility an-

chor that makes sure packets are routed properly to then be delivered between the UE and PDN even though the UEs position changes. Finally, the User Plane (UP) includes mobile anchor functionality, external PDU session connection points, and packet routing and forwarding. This is where the packet traffic is carried over the data-link layer using Ethernet-frames with MAC-addresses, which is where Network Interface Controllers (NICs) are used [22, 23, 24, 16].

The cloud infrastructure layer contains all cloud functionality and resources including such for security. These are all the tools needed to build the cloud. Finally, the transfer layer contains the hardware for the network communication [16, 12].

2.2 Smart Network Interface Controllers

This Section vows to give a conviction to the need for Smart-NICs in the 5G industry. Following is an explanation to how a Smart-NIC works in more detail.

2.2.1 The NIC

A NIC is a hardware component used to connect a computer to a network. It acts as a packet transceiver and has as a main function to convert packets between digital signals and data form. The NIC provides devices with MAC addresses and strips or appends the packets' data layer frames upon arrival and dispatch respectively [25].

General NICs contain multiple smaller parallel cores in order to handle multiple transmit/receive queues in parallel as well as increase packet processing speed [6]. Older NICs may further use the host's general use CPU for running network stacks. However, the use of host CPUs for running networks stacks in the time of cloud computing has been proven to decrease available processing power from VMs and increases latency to the network performance [26]. Therefore, modern NICs contain extended hardware in order to offload the CPU on top of optimization hardware to increase network packet processing speeds [9]. For example, NICs may have offload engines (such as the TCP offload engine [27]) to relieve the TCP/IP packet processing stack from the CPU and include interrupt and Direct Memory Access (DMA) interfaces to the host processor to avoid accesses needing to pass through the CPU [9, 28].

Low speed (<10Gbps) packet processing is still possible using simply Operating System (OS) drivers paired with high-end CPUs. However, the mentioned extended techniques are crucial to reach speeds of 100G without overloading the CPU [9].

2.2.2 The Need for Smart-NICs

While the regular NICs provide several performance increasing features, problems arrive when NICs are used in the continuously evolving networks of today. Firstly, the networks' applications demand increasing amounts of features including support for new protocols, evolving intrusion detection, and dynamic load balancing, on top of the already existing ones. Secondly, the rapid development of 5G has created the

possibility for new network use cases such as cloud computing, machine learning, and big data applications. As a result of this there is a need for new ways to architect communication networks.

The requirements of flexibility, dynamicity, performance, and efficiency demand carefully engineered NICs, which not only require a lot of time and money to develop, but also large deployments of dedicated hardware to run. Due to these issues, vendors tend to only add features which have been in demand for a longer time, which slows down development at the same time as the further addition of features slows the NICs down [29].

To mitigate the gap between the increased network bandwidth and the stagnating computing power of the CPUs as well as mend the flexibility issue of the NIC, SNICs have been developed [8]. Smart-NICs are network interface cards that provide programmability in the data path [6] by extending the foundational NIC with a programmable engine such as a Field Programmable Gate Arrays (FPGA), Application-Specific Integrated Circuits (ASIC), or an embedded CPU. This engine makes it possible for new features to be introduced to the already existing SNIC by using the programmable hardware functions [30].

Due to its programmability, the introduction of SNIC makes it possible to use SDN programmable data and control planes. By using these SDN planes, the features of the packet processing and routing can be defined in software and then optimized in hardware [9]. The programmability also makes the SNIC able to accelerate a larger variety of workloads compared to the previous ones [31].

The use of programmable SNIC, to offload logic for packet processing has already been researched and proven to improve processing speed even while using only partial offload [5, 6]. The research of the possibilities of offloading network stack functionality onto the SNICs has lead to development of multiple different system models designed for ease of use and performance in different environments [7]. For example, there are frameworks specialised in offloading distributed applications [8] and for running Azure network stacks [26].

Though the programming models of the Smart-NICs still are in development, there are languages, such as P4, that are designed with the features of Smart-NICs considered [6].

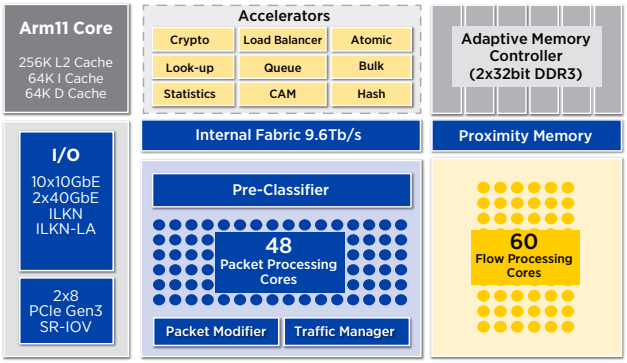

Figure 2.4: NFP-4000 Flow Processor Block Diagram [32].

2.2.3 Smart-NIC Architecture

To be able to add support for new functions to be added after purchase, the Smart-NIC needs additional computational power and onboard memory which is not provided to a regular NIC. These needs can be provided by different architectures, but most are constructed with a control processor at the core [33]. This core is used to initialize and configure the SNIC, hold all additional features, as well as other control plane management features such as packet steering. By using this additional core, the host CPU's use of cycles is offloaded [34].

The additional computational power is usually provided by adding clusters of ASICs, Flow Processing Cores (FPCs), or by using an FPGA. Where ASICs are circuits specialized for a certain purpose, FPCs are customizable cores designed for optimizing packet flow processing, and FPGAs contain low level reconfigurable circuits. FPCs provide the cheapest, most flexible solution in comparison to the other two and is hence the most popular architectural approach [35, 33].

2.2.3.1 Netronome Agilio Smart-NIC

In this project, a Netronome Agilio SmartNIC with a Netronome Flow Processor (NFP) 4000 as control processor will be used. These NFPs use the FPC approach, supporting multiple threads each with access to dedicated instruction and data memories [36]. The Agilio NFP-4000 has possiblity to support up to 60 FPCs with each FPC able to use 8 threads each. This makes it possible for the SNIC to process 480 packets in parallel. A closer look into the NFP can be seen in Figure 2.4, where the output PCIe-Gen3 interfaces as well as different accelerators are visible. A sample design of a SmartNIC can further be viewed in Figure 2.5 where it is connected to a host through the PCIe-Gen3 interfaces. This section will now continue to explain a packet's path through the Agilio NFP-4000 SNIC.

When a packet is received at the SNIC ingress port, it firstly goes through an integrity check. Thereafter, it is stored in a buffer and delivered to the ingress processing stage. In the processing stage, Packet Processing Cores (PPCs) are used to parse the packet, generate its metadata, and finally the packet is sent to the DMA engine. The DMA engine then sends the packet headers to the FPCs and the

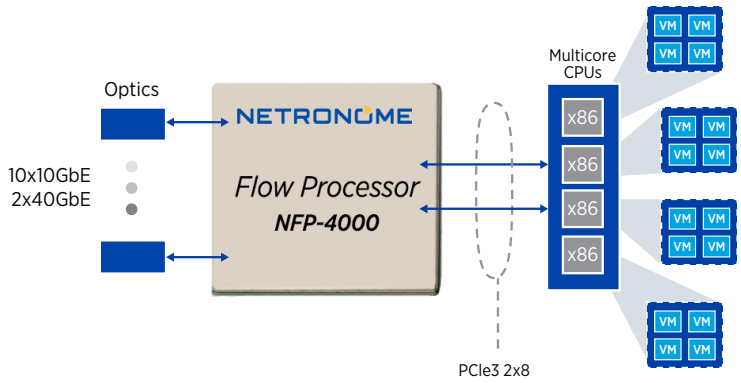

Figure 2.5: Agilio SNIC Sample Design [32].

payload to a buffer. All of this is done under complete software control. The FPCs do the flow processing, including generating keys for table matching, table lookups, and forward, Drop, and Add/Remove header actions. Finally, the PCI repackages the packet with headers and payload, and sends it to the host. When the packet is delivered from the host to the egress port instead, the PCI first delivers the packet to the FPCs where packet processing is performed. Thereafter, the packet is sent to the Egress PPCs where it is put in a traffic queue, to finally get its checksum computed and be delivered to a network interface the Egress MAC.

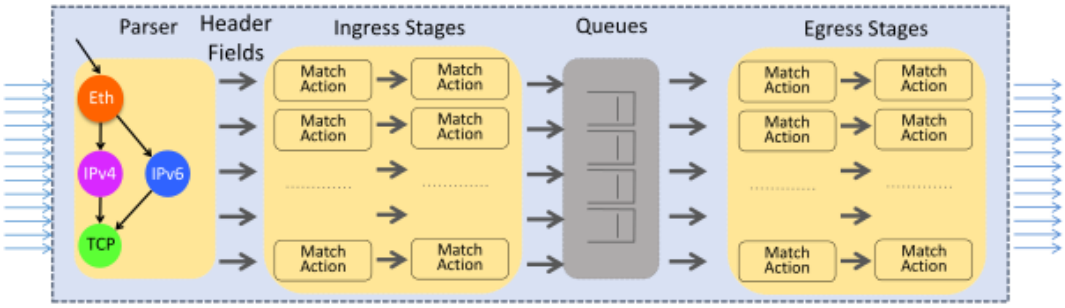

Figure 2.6: The P4 abstract switch model [38].

2.3 The Packet Processing Pipeline

The packet processing pipeline contains multiple stages performing different network functionality that may change depending on what functionality is needed. When a packet arrives a typical pipeline may begin by extracting necessary data from a packet. In this step, multiple headers may be parsed and data from them can be used to match against a set of rules decide if the packet should for example be dropped, delivered, or maybe a reply should be sent. The next step could be a security check, to see so that for example headers are formatted in the way they should be, that they have the correct sequence numbers, and that the packet isn't expired. A further step could be to change certain values of the header, for example to decrement the TTL. The final step could be to perform some kind of action with the packet, for example send a response, or to reassemble the packet and change its address to the next one in the switching table. A NIC can be used to perform some or all of these actions. With the SNIC, it's possible to change the actions on demand, while the NIC needs physical upgrades to adapt to new actions.

2.4 Introduction to the P4 Language

P4 is a high level language for programming protocol independent packet processors, with three main goals in focus: reconfigurability, protocol, and target independence. Because of these properties, it is well suited to deal with some of the challenges of adapting to the requirements of 5G and its multitude of IoT devices. While the programming framework of P4 is not the standard for SNIC programming, the fact that P4 is protocol-independent and can be compiled and run on multiple different targets (FPGA, CPU, ASIC) [37] makes it a suitable language choice for SNICs. P4 programs are able to run on systems with different vendors and architectures, easing the integration process into preexisting systems. P4 programmed SNICs have been proven to increase flexibility of packet systems [9].

The P4 model, as shown in Figure 2.6, generalizes how the processing works in different forwarding devices and by different technologies. It is this generalization that makes P4 programs special. The different parts of the model, from the header fields and parser to the actions in the match+action stages, are all configurable by

```
1  header ethernet_t {
2      bit<48> dst;
3      bit<48> src;
4      bit<16> etherType; }
5
6  header ipv4_t{
7    // field definitions
8  }
9
10 header some_header_t {
11     bit<32> some_type;    }
12
13 struct headers {
14     ethernet_t ethernet;
15     ipv4_t ipv4;
16     some_header_t some_header; }
17
18 struct metadata {
19     bit<32> useful_information;
20 }
```

Listing 2.1: P4 header example.

the programmer. In the following sections of this chapter the different parts of a P4 program will be further explained.

2.4.1 Headers

In P4, headers must be explicitly defined by the programmer. As shown in the example in Listing 2.1, each header and the fields it contains must be stated in order for the program to recognise it in a packet. The definition includes every field in the header and their length in bits. As seen in the example, the programmer can construct headers freely, with fields of varying sizes.

The different headers can be used to make header structs, that are used in the program. The header struct on line 13 in Listing 2.1 states the three headers that may be present in the incoming packets. This struct is then passed through the different parts of the program, and the stated headers, if present, can be accessed and modified. Not all headers in the struct must be present in the packet, but any headers that are extracted by the parser must be part of the header struct. On line 1 in Listing 2.2, the output of the parser is signified by the keyword `out`, in this case `hdr`.

Aside from the packet headers, metadata can also be stored and accessed throughout the processing. In the parser definition, as well as in the definition for every control block, the metadata is both input and output, using the `inout` keyword. The structure of the metadata is defined in the metadata struct on line 18 in Listing 2.1. In the example, there is one 32 bit field in the metadata, but it is up to the programmer to define the number of metadata fields and their respective length.

```
1  parser MyParser(packet_in packet, out headers hdr, inout metadata meta, inout
       standard_metadata_t standard_metadata) {
2
3      state start {
4          transition parse_ethernet;
5      }
6
7      state parse_ethernet {
8          packet.extract(hdr.ethernet);
9          transition select(hdr.ethernet.etherType) {
10             TYPE_IPV4: parse_ipv4;
11             default: accept;
12         }
13     }
14
15     state parse_ipv4{
16         packet.extract(hdr.ipv4);
17         transition select(hdr.ipv4.protocol){
18             PROTOCOL_SOME_PROTOCOL: parse_some_header;
19             default: accept;
20         }
21     }
22
23     state parse_some_header{
24         packet.extract(hdr.some_header);
25         transition accept;
26     }
27 }
```

Listing 2.2: P4 parser example.

2.4.2 Parser

The parser ensures that the header sequence of incoming packets are valid and extracts the values of the header fields. As seen to the left in Figure 2.6, the parser is constructed as a state machine traversing the header fields and creating a parsed representation of the packet to be used in the processing.

At each state, the parser can extract headers, and check the values of any extracted fields to determine the next step. The first state of the parse is called `start`. In the example in Listing 2.2, this step immediately moves to extract the Ethernet header, expecting it to be present in any arriving packet. Depending on a value in the header, the next state subsequently is determined. Parsing is completed when transitioning to the `accept` or `reject` state.

The state machine functionality provides a means for P4 programs compiled to programmable switches, to handle new protocols without the need for individually reprogramming the switch or getting specialised hardware. During parsing, as well as later in the process, metadata can be added to the P4 packet to be used in the processing. A short snippet of a P4 packet parser is included in Listing 2.2. The parser definition includes the type of header structure the parser will produce, in this case `headers`, as defined on line 13 in Listing 2.1.

To later be able to reassemble the packet before forwarding, there is also a deparser. The function of this part of the program is to take the parsed representation of the

packet, with all its metadata and state which parts are to be present in the outgoing packet, deparsing the relevant headers and discarding all metadata. The deparser uses an inbuilt function in P4 to emit packet headers in an order specified by the programmer, an shown in the example in Listing 2.3. Unlike the parser, the deparser is not a unique structure in the P4 program, but rather it is the last of the Control Blocks, further described below.

```
1 control MyDeparser(packet_out packet, in headers hdr) {
2     apply {
3         packet.emit(hdr.ethernet);
4         packet.emit(hdr.ipv4);
5         packet.emit(hdr.some_header);
6     }
7 }
```

Listing 2.3: P4 deparser example.

2.4.3 Control Blocks

The control blocks of a P4 program specifies what will happen to a packet after it is parsed. Depending on what headers are present in the packet, or other data extracted by the parser, it determines what actions and tables to apply to the packet. It resembles an imperative program, made up of conditionals, functions and references to tables and actions, but without any loops.

There are usually at least two control blocks in a P4 program, one for ingress and one for egress processing. Each control block can contain multiple actions and tables, that can be referenced within the block. Aside from action and table definitions, the control blocks main function is called `apply`, and it is this function that is called as the control block is executed. An example of a control block is shown in Listing 2.4. The execution starts on line 14, where the program first checks if a certain header was present in the incoming packet. If so, the action `packet_forward` is executed. Otherwise the packet is dropped.

2.4.4 Actions

Actions are constructed from a set of some predefined actions, for example adding and removing headers or modifying field values. Shown on line 7 in Listing 2.4 is an action, `packet_forward`, that replaces the destination MAC address with the source in the ethernet header, and sets a value in another header to a predefined value. Actions can be defined within control blocks or on their own. An action defined within a block can access any input to that block, stated on line 3 in the example. The `packet_forward` action in this case is directly changing values in `hdr`. While this is useful in many cases, it might not always be preferable as if an action is defined within a control block, it can only be accessed from within the same block.

```
1
2  control MyIngress(inout headers hdr, inout metadata meta, inout standard_metadata_t
       standard_metadata){
3      action drop() {
4          mark_to_drop(standard_metadata);
5      }
6      action packet_forward(){
7          /* Swap the MAC addresses */
8          bit<48> tmp;
9          tmp = hdr.ethernet.dst;
10         hdr.ethernet.dst = hdr.ethernet.src;
11         hdr.ethernet.src = tmp;
12     }
13     apply {
14         if(hdr.header.isValid()){
15             packet_forward();
16         } else {
17             drop();
18         }
19     }
20 }
```

Listing 2.4: P4 control block example.

```
1  action set_some_type(bit<32> some_type){
2      meta.some_type = some_type;
3  }
4
5  table some_type {
6      key = {
7          hdr.ipv4.dst:  lpm;
8      }
9      actions = {
10         drop;
11         set_some_type;
12     }
13     default_action = drop();
14 }
```

Listing 2.5: P4 table example.

2.4.5 Match+Action Tables

The match+action tables are used when information not necessarily available at compilation is needed to determine the actions to take in the processing. An example in shown in Listing 2.5. In this example, if the destination IP address is present in the table, the input value for `set_some_type` will be provided by the table, and then the action preformed. The keys of a table can be matched either exactly, using wildcards, or as in this case, using `lpm`, the longest prefix match. The possible actions when a match occurs are stated in the table specification, as well as the type of matching. If there is no match, the specified default action will be taken.

The size of the table can be specified explicitly, as in the example, or left up to the compiler to determine.

```
1 extern Register<T> {
2     Register(bit<32> size);
3     T read(bit<32> index);
4     void write(bit<32> index, T value);
5 }
6 Register<bit<32>>(1) count;
```

Listing 2.6: P4 extern register definition [39].

2.4.6 Externs

Externs have many uses in P4, such as extern blocks and functions, however in this section we will look closer at one example specifically relevant to the project, namely as a way to implement global objects that are shared between executions. Extern objects rely on the underlying architecture functions to implement things like registers in P4 programs.

An example of an extern definition is shown in Listing 2.6. The definition of the register object is much like an abstract class in object oriented languages, it states the constructor and methods of the object, while leaving the implementation of them unspecified. This particular extern takes a type and a size, and represents a register that is in effect a list of variables of that type. The definition is then used as a type, and objects of that type can be initiated as shown on line 6 in the example. This register `count` is then shared between the threads and executions of the program.

2.4.7 Compilation

The compilation of a P4 program is done in parts, each part shown as a box in Figure 2.7, where yellow represents a representation and blue a part of the compiler. First, the program's Intermediate Representation (IR) is generate by a front-end compiler, such as the open source GitHub compiler by p4lang [40]. After this a target specific compiler is needed to convert the code to code functional for the target device. In the case of the Netronome Agilio SmartNIC, the back-end compiler converted the IR to C for the data path on the NFP. The Netronome SDK then uses the Netronome Flow C Compiler (NFCC) to be able to compile and the C code to generate the NFP firmware file, which can then be uploaded to the SNIC using Netronome's own uploading software. Tables are written as JSON files, either manually or by using Netronome's Programmer Studio, and are uploaded to the SNIC thorugh the Netronome Runtime API.

2.4.8 Limitations of P4

There exists several limitations in the P4 language, following, the one's relevant to this thesis will be presented. To begin, there are no pointers in the P4 language, however this is not unusual for programming languages. Neither are there any switch statements, which instead can be implemented as match action tables. Return statements that makes the code return in the middle of an actions does not exist either.

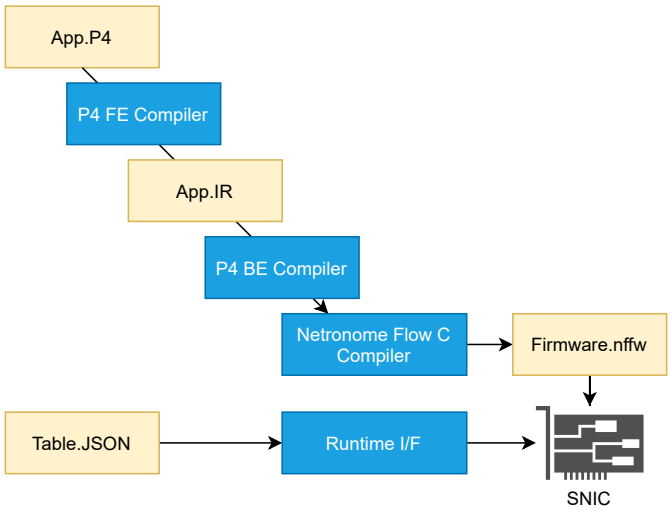

Figure 2.7: The P4 to Netronome Agilio NFP-4000 compilation.

There are no arrays in P4, which makes it necessary to implement each array element as an own variable, or perhaps as one variable containing a multiple of the number of bits of the elements' size. There is also no way to accept headers with variable length-field unless multiples of such a header with different length of such fields is defined. Hence, application layer information is difficult to inspect, since these usually differs greatly.

The lack of arrays, is however not a big loss since the P4 language does not contain any loop-functionality for processing the arrays anyway. Any loop functionality will need to be rolled out in some way, and the number of times the loop is performed has to be hardcoded (if-statements could be used for certain cases).

Finally, there are no variables saved between different packets unless registers are created and values are stored in these. Though it is not certain that all Smart-NICs have support for such registers.

It is worth noting that there is support for using external C-code in some Smart-NICs. Which may be a way to circumvent the issue of these limitations, but due to the project striving for the most general implementation which should work on various Smart-NICs, as well as optimal performance, no deeper investigation of external C-code was performed.

2.4.8.1 Differences Between $P4_{14}$ and $P4_{16}$

The first version of P4, now referred to as $P4_{14}$, was released in 2014. Following the release the language was updated frequently to meet the demands of the growing user base. In order to keep up with the expansion, the language went through several significant changes in it's earlier years. In an attempt to stabilize the language while maintaining the ability to adapt and include new functionality when necessary, $P4_{16}$ was released in 2016. $P4_{16}$ split what was $P4_{14}$ into a smaller language and a core library, namely `core.p4`, containing some fundamental constructs. While the

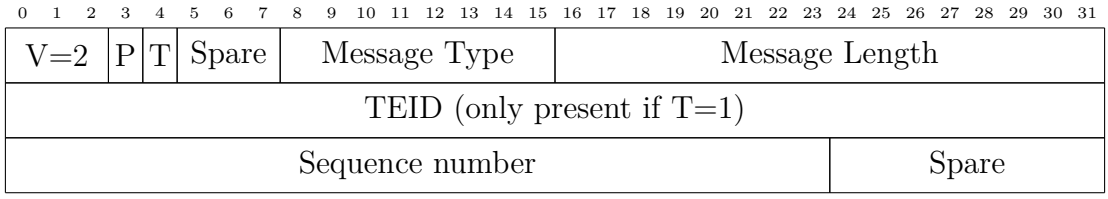

Figure 2.8: The GTPv2 Header.

versions are similar and the syntax looks much the same it is important to note that $P4_{16}$ made many backwards incompatible changes from $P4_{14}$.

One main difference regards how there are no control blocks in $P4_{14}$, where all actions and parser pieces are placed directly in the file with different prefixes. This difference makes compiling $P4_{14}$ in a $P4_{16}$ compiler not possible and neither the other way around. Another significant difference between the two versions lies in the way that $P4_{14}$ does not use a deparser but simply emits the packet headers as they came in. This results in a severe lack of flexibility in $P4_{14}$, since headers can only be modified and not removed or added.

2.5 The GPRS Tunneling Protocol

The GPRS Tunneling Protocol (GTP) (shown in Figure 2.8) is one of the protocol that the chosen part of the system was programmed to handle and therefore needs to be briefly presented. GTP is a protocol used by mobile network operators on interfaces within Random Access Networks (RAN), roaming, and in the packet core. The protocol allows mobile device users to be continuously connected to the network while moving [41].

To connect through GTP-Interfaces and to divide traffic into different communication flows, GTP uses tunnels. These tunnels transport IP-payloads between different user and GTP equipment and the Internet. These tunnels are then modified, removed, and added on demand. The protocol focused on in this projects is the user plane protocol GTP-U, which handles user data.

2.6 Software Tools

During this project, a collection of software tools have been used. The first two sections describes Scapy and Mininet which were involved in the initial testing of the P4 code. Section 2.6.3 then describes the Agilio P4C Software Development Kit which was acquired after the SmartNIC was ordered and primarily used for testing using the SNIC. Finally Section 2.6.4 and 2.6.5 introduces Docker and the Data Plane Development Kit (DPDK), used in the testing environment.

2.6.1 Scapy

Scapy is a powerful and versatile tool for network manipulation written in Python [42]. Scapy can be used for a wide range of purposes including scanning, fingerprinting, unit testing, attacking, sniffing. For the purposes of this project, however, the focus was Scapy's incredibly flexible packet forging. It enables the user to create packets freely, defining new headers and inserting any values in the fields of existing ones. The headers can be layered in any manner the programmer choose without limitations of predefined templates or methods. Scapy is often simple to use as it is possible to build almost any tool you could think to require, and most of the time, it can be implemented in just two lines of code [42].

2.6.2 Mininet

Mininet [43] is a tool for creating a virtual network and is useful for testing and developing SDN functionality. Mininet can simulate multiple hosts and switches on a single machine, with the code executed at the nodes running on a real Linux kernel. This allows the user to run any program they wish to control their network. Using Mininet, the user is able to easily create a network with a desired topology. This network will be a realistic simulation without the need for any additional hardware. This means that the code developed in the simulated network can be used in a physical network with little or no modification needed.

2.6.3 Agilio P4C Software Development Kit

The Agilio P4C Software Development Kit (SDK) 6.x is a datapath programming tool which supports C and the P4 language. It contains a Full-Featured IDE for windows called Programmer Studio, which includes a GUI and simulator. The simulator is however deactivated in Programmer Studio 6.1.0, which is the only version supporting $P4_{16}$, leaving it unusable in this project [44].

Further contained within the SDK is also the Linux and Debian back-end toolchains containing the compiler, assembler, and linker. This toolchain maps P4 code to any Netronome flow processor. The SDK host software's also contains a real-time environment which includes functionality to update the NICs rule tables at run-time as well as a means for the SDK to download user-programmed binaries to the SmartNIC.

2.6.4 Docker

Docker [45] is a an open source project that lets you design, build and run software in an isolated container. This container can be constructed to contain any dependencies you require, and the behaviour inside the docker container is the same regardless of the underlying device. While similar in many ways to a VM, a docker container is much lighter, as it does not require an entire OS to run, only the bare necessities are included.

2.6.5 DPDK and pktgen

The Data Plane Development Kit [46] is a set of libraries aimed at enabling fast and efficient packet processing. It is a vast open source project with many academic and industry contributors and users, with support for many if not all major CPU architectures and NICs. DPDK can be used for many things and contain a vast array of functionality for multi core packet processing in a cloud infrastructure. The Poll Mode Driver (PMD), is a DPDK feature that enables the application to use as the name suggests, a poll mode rather than interrupt to receive packets, making for highly efficient transmission. Powered by DPDK, pktgen is a powerful packet generator that can generate highly customizeable traffic flows at line rate. Pktgen can generate a large amount of packets that are created on the fly using a user provided configuration and TCP/UDP port numbers. The software selects the packet to create depending on which UDP/TCP port it is addressed to, which can be selected as a range of multiple ports by the user. Then the user provided range of addresses (Ethernet and IP) is used.

2.7 Related Work

The testing done in many previous papers have been executed on systems created for the purpose of assessing gains of using SNICs in packet processing scenarios only in controlled testing environments [5, 6, 7, 8, 9, 10]. Other papers instead only regard integration into a special kind of system [26]. However, there has been little research of how to integrate the P4 programmable SNICs in deployed industrial systems. This is something in need of research for the SNICs to be applied to real scenarios. The remainder of this section will present three papers relating to the integration SNICs in modern networks.

2.7.1 Offloading 5G Functions to a Programmable ASIC

Promising research has been done in regards to offloading user plane functions using P4. In [47] the authors explored the possibility of offloading the 5G Mobile Packet Core (MPC) User Plane Functions to a P4 programmable switch ASIC. In the project, a Virtual Evolved Packet Gateway (vEPG) [48] pipeline was implemented in P4 and compiled to a Tofino Barefoot programmable ASIC.

The research shown that the Barefoot Tofino switch can handle the vEPG functions at the rate they are received with low latency. The tests performed in the project were with up to 100 000 active users, but the authors theorize that the results could scale up to accommodate as many as 1.7 million in a commercial switch. The authors tried variations of optimising the P4 program with significant changes in the performance and memory use, showing that P4, like many other languages, can be optimised with great results.

The research in [47] much like that of this thesis concerns offloading in a 5G network with the use of a P4 programmed switch, a main difference being the use of a programmable ASIC rather than a SNIC.

2.7.2 Using Smart-NICs to Decrease Host Processor Usage

In [5], an approach for effective integration of SNICs as a way to decrease host processor usage while not stressing the SDN-controller is presented. The paper motivates the need of SNICs to relieve the host processor of NFs. Further, it discusses the issue of using SNICs for NF offload as a separate entities from the host hypervisor to the external network due to the increase in management pressure this puts on the external SDN-controller. To solve this issue, a generalized SDN-controlled NF offload architecture, UNO, is suggested. This architecture makes it possible to hide the existence of SNICs in the local network behind a virtual plane so that no changes in the external planes need to be made.

The UNO virtual plane is responsible for mapping virtual switches, shown to the outside with the physical switches divided over several SNICs and hosts. It also contains a translation algorithm for rules gained from the SDN-controller to "virtual-rules", due to that a virtual egress and ingress port can be mapped to two different switches.

Since multiple NFs interact with each other, the placement of the NFs within the local switch network is proven important. To minimize PCI-bus usage (which increases latency and decreases throughput) the UNO virtual control plane therefore provides an algorithm for determining which switch a NF should be placed for optimal performance.

The benefits of the UNO-architecture using a partial offload algorithm is lower CPU-usage, lower power usage, higher throughput, and lower latency than when using only a host processor. Further, the research proves partial offload to give better latency than when applying full offload.

This study suggests a plan for integrating Smart-NICs as a stand-alone entity in SDNs, refraining from using the host processor as a gateway. However, the process does not involve how to use the concept in practicality. If our research shows frutiful however, trying to apply the UNO functionality could be a further step in decreasing load on the host processor.

2.7.3 P4-enabled Smart-NIC Offloading

To cope with the requirements of 5G, the authors of [9] investigate how offloading data plane functions to a Smart-NIC decrease latency and increase the throughput of a network. The authors describe the process of design, implementation and validation of a P4 programmed FPGA Smart-NIC. They also show how this solution improves the flexibility of the network, enabling the functionality of the data plane to be changed in seconds.

In order to optimize the performance of the setup, packets are processed according to their specific requirements, whether it be low latency or throughput. Segment Routing (SR) [49], which uses a special header to redirect data along a chosen path, is used to make this functionality work. P4's ability to insert specific headers is used

to help the processing in the beginning of the pipeline, and removing them in the end.

The Smart-NIC was able to reach an impressive maximum throughput of 84.82 Gbps and the bandwidth of the network reached up to 30% wider with compared to without the SNIC.

The focus of [9] closely resembles that of this thesis in the use of a P4 programmed switch for handling NF to meet 5G requirements. Some differences in [9] are the focus on network slicing specifically as well as the FPGA-based SNIC.

3

Implementation Methodology

In this chapter the process of implementing the P4 program is described. First, the development environment used will be presented, and then the different steps of developing the program for the SNIC will be discussed. Finally, the integration of the SNIC with the industry pipeline is explained.

3.1 Environment Setup

As part of the resources available on GitHub from the P4 consortium, there was tutorial to the language. This tutorial includes multiple exercises designed to let the learner implement missing parts of different P4 programs. Since these exercises were made for a workshop setting, there also existed a downloadable virtual machine set up with the P4 compiler and all its dependencies to be able to program directly. The virtual machines also contained an automatic Mininet setup to run a specified topology every time the project was built. This mininet setup uses the P4 compiler's backend support for the library p4c-bmv2, which is the behavioral model used for setting up virtual software switches.

Scapy was also already installed on the VM, together with Python code for sending packets from one host to the SNIC and then receiving the outgoing packet on another host. Using this Mininet-Scapy setup, it was easy to make sure that the code functioned properly.

In the exercises, the switch programming model provided was the v1model, shown in Figure 3.1, which is a library specified to work with the virtual switches defined by the bmv2 backend. Due to its general approach, and direct support by the p4c compiler, the guess was that few, if any, changes would be needed for the code created for the v1model to work on the Agilio SNIC.

Completing the exercises on the virtual machine was a good introduction to P4 as well as to the Mininet setup. The provided environment could also easily be used to develop new P4 programs. The automatic Scapy-Mininet setup was convenient, and easily configurable. This was a time-saver, since setting up the P4 language from scratch in Windows (which was the OS installed on the company computers) was especially complicated (and not fully supported).

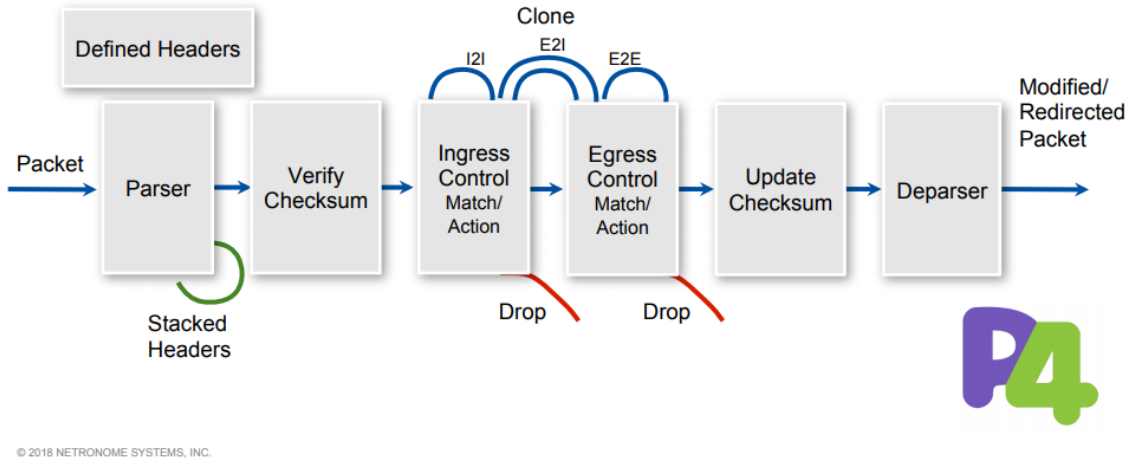

Figure 3.1: The v1model [50].

Using the same Mininet environment, it was also possible to use C rather than Python to send and receive the packets at the hosts. This was useful to further on be able to develop test programs in C to more accurately imitate the original code and to easily send and receive packets with custom headers needed for the project.

3.2 Initial P4 Test Program

The initial P4 program created was made by taking bits of pieces of the provided exercises to first be able to send a packet from a host to itself. The recycling of parts ensured the new P4 program was built according to the v1model standard and simple to test using the provided Mininet topology. A new header was created to be able to receive input from the existing system code. The header is placed underneath the IP-protocol and the python code for Scapy was changed to support this. To ensure the program was working correctly, the fields of this header was changed during P4 processing. After making sure the packet with added header was processed and received properly, the existing system code to be translated could be examined.

3.3 Translation of industry code to P4

This Section aims to present the functionality which the program fulfills and discuss the changes that are needed to translate C-code to P4.

In consultation with an industry expert, a small part of the program was isolated. The purpose of this part of the program is to perform the initial classification of incoming packets and calculate some values to be used at a later stage. The target code classifies two types of incoming packets, regular IPv4 packets and GTP IPv4 packets. This part was chosen due to its relatively isolated functionality which does not depend on many other files.

3.3.1 Brief Overview of The Original Program

In the program, a packet arrives on the ingress port and, after calling some intermediary functions, calls the classify function. The classify function then classifies the packet as depending on its destination address and certain current system variables. The program then calculates the relevant values for the classification of the packet, puts the values in a struct, and returns it. The result struct is then used to set the packet structs header offsets and further its values are used when the packet is processed to be forwarded or dropped.

3.3.2 Required Changes for Translating C Code to P4

The packets that arrive to the SNIC are received from the external network, which means that ethernet headers are present in the P4 program, while the pointer to the packet struct that is delivered to the packet-classifying function already has the ethernet header removed. However, this does not affect the functionality of the program.

In the packet processing using C language, pointers are frequently used to point to different parts of a packet, such as headers and fields. However, this approach is not needed in P4 where the parser and header definitions are used to extract all headers and their fields before performing any functions.

Generally, C program functions also use pointers to structs to deliver the necessary values to other parts of the code. This is not possible when using the SNIC which can not communicate directly with the code but instead must send the information through packets. The solution to this problem is to, instead of using structs, create a new headers and add these to the packet upon exiting the SNIC. This also means that the point where the packet is delivered will need to extract the values of this header to then put in a struct to deliver to the rest of the code.

To be able to save variable values between runs, P4 uses registers. These registers are present only if the SNIC has capacity for them. The values in the registers are then updated by sending certain control messages to the SNIC when the system value is changed. To do this, a custom configuration header need to be added and the protocol needs to be changed to indicate this. This header contains the new values and a flag field, to indicate which values to update. Values that are temporary (per packet) are simply saved in metadata until the headers' values are set in the deparser.

Usually, packet classifiers also contains lookup tables and switch statements which in P4, can be done using a match+action tables. A switch statement may simply use a variable as input to a table and an action is selected depending on the input value. For a lookup table, the method would be a method that sets a specific variable in the metadata, which may then be treated as the looked-up variable.

Arrays may also be used, which is neither applicable in P4. In some cases, this could easily be solved by replacing the array with separate variables or merging multiple

variables into a single bitfield using the OR operator and bitshifting. However, for larger arrays, it is possible a different solution, containing external code in C when the need for arrays occurs, could be applied instead. The netronome SNICs used in this project supports this functionality of using external C code, it is however uncertain whether all SNICs allow this.

Finally, a regular C-program usually contains some kind of for or while-loops, which do not exist in P4, a language which does not contain any loop functionality at all (except in the parser where there are GOTO statements). If the number of loops is predictable, the solution could simply be to create a hardcoded implementation by using loop unrolling. Another solution to loops could also be external C code, as mentioned above.

3.3.3 Difficulties in Using Netronome's Programmer Studio

In order to eventually compile and run the code on the SNIC, the program needs to be built using Netronomes SDK on Linux or with Netronome's Programmer Studio IDE on Windows. The IDE, which was used for ease of setup, comes with certain difficulties that can be hard to predict. The error messages provided by the IDE are at times problematic to interpret and often lie in the so called list files generated during the build process. We personally believe that some form of bug is involved, as standard P4 keywords cause errors to occur whenever used, which is improbable to be intentional. It is also the case that some programs will compile correctly once, and cause errors the next time, despite no changes having been made to them.

One method of combating this behaviour used in the project was to simple insert the code in pieces, and rewrite anything causing errors to circumvent them. The IDE cannot handle the `return` statement, and so they need to be eliminated and the code adapted accordingly. The inbuilt function `isValid` is another unusable one in Programmer Studio IDE. The `isValid` function is a function which can be called to check if a certain header was successfully extracted using the checksum and whether or not the header was parsed at all (aka in the packet). In order to preserve the functionality of the program, metadata fields to for each header can be set in while parsing that can then be used in the processing. Of course, this is a breach of security since this would cause even a deformed header to be seen as valid, making the system open to for example DDOS-attacks. However, security was not of concern to this project. It would also be a possibility to check the checksum manually if security was of interest.

3.3.4 Dynamic Table Updates

A further necessary feature of the program is to be able to update the tables, without recompiling the program. In this projects case, Netronome has its own CLI already set up to perform this feature.

While Netronome SNICs have support for table updates through the described CLI, it is not certain whether all SNICs have such features available.

3.4 Creating the C-Program for Testing

The SNIC, as previously mentioned, is in this program the link between the external and internal network. This is different from the way it is programmed in the original program, where a separate packet receiver is implemented in another file and only pointers to the packets are sent to the classifier. Hence, there is need to supplement some C-code to receive the classified packet to then mediate data found in the result header as well as the packets location in memory to the rest of the system. Further, to be able to use the program separately from the rest of the original environment, several struct and methods from the original program need to be included.

The following subsection presents how the C-program was implemented as a standalone program, followed by a subsection describing adaptions needed to the original systems to use the new packet type as provided by the SNIC.

3.4.1 Adapting the Code for Standalone Tests

The C-program's standalone version was first implemented for Mininet testing and used the netinet library to receive and pick out the different layers of the packet. The packet's values were then checked so that packets arrived correctly with correct values after going through the P4-program on the SNIC. Some values were for example calculated beforehand and then compared to what the SmartNIC delivered in its added header.

For testing within the testing environment the complete classifer was extracted from the original code and then a process to pick out necessary includes and remove the code not necessary was conducted. Since the program, used as a basis for performance testing measurement, already had a way to create a pointer to a packet, it was possible to let it function as the original code by only supplying the classifer with a struct pointer, the pointer to the packet, and the packet's length. Hence, the original receiver created was not used in the actual testing process.

3.4.2 Adapting the Original System

In the original system, one method is used to do all classification of the packet. However, when using an additional header, this kind of method is no longer needed. Instead, the classification method should be replaced with a converter method which takes the values from the added header and puts them in the respective field in the struct. It must also add to the struct any values that are not present in the added header, such as system variables and the packet's memory address.

Some variables in the original system, are not optimal to save in registers within the SNIC, since these values are easily accessed from within the original system. These fields of the struct should preferably instead be set somewhere within the original system instead of being picked from the result struct. However, in our implementation, we provide a register solution to prove that it is possible to provide the values from the SNIC as well. For this to work properly, the original system will

need to send out updates with the register change protocols to the SNIC whenever these values are changed.

3.5 Necessary Downgrades of the Program

Apart from issues occurring when trying to compile the program with the Netronome SDK, more difficulties were faced when trying to load generated firmware onto the SNIC. These issues whose reasons are still unknown, caused the Netronome loading software to not be able to read the program configuration files containing program tables. After weeks of being in touch with experts in using Netronome SNICs and Netronome's Support (who unfortunately didn't have time to look into the issue) we decided the only way was to not use tables. This meant that any necessary dynamic functionality would need to be hardcoded.

Due to the time loss, we also decided to refrain from examining the use of registers, since we would not have the time to investigate a way to properly use these.

Due to previously mentioned removal of tables, the use of the dynamic tables using CLI could not be investigated further.

4

Evaluation Methodology

This chapter presents the methodology used for evaluating of the results. The methodology includes a performance evaluation section and a qualitative section. The former giving insight in how the implementation using P4 performs compared to the initial C implementation and the latter comparing different non-performance related variables concerning the two implementations, such as the difficulty of programming in the two languages.

4.1 Performance Evaluation

In order to compare the performance of the initial C implementation of the system section to the translated P4 version, it was first necessary to decide how to measure their performances. The sponsoring company strongly suggested packet throughput, which was their main performance measurements, and after looking at other works within the area, it seemed to be worth focusing on and then take other measurements if time allowed (it did not). The test environment was to be set up between two machines. One would run a traffic generator and the other one would run a test program and by receiving the traffic measuring the selected measurement both for the C program, with a program letting all packets through on the SNIC, and the P4-program.

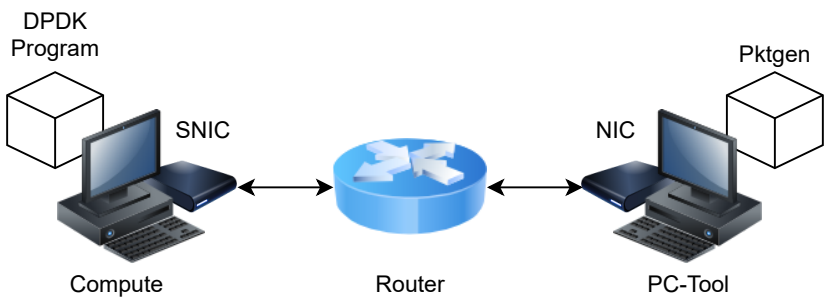

Figure 4.1: Performance evaluation setup, the cubes representing the hypervisors.

4.2 Test Environment

The basic testing environment, shown in Figure 4.1 consisted of two machines; "Compute", which was connected to the SNIC, and "PC-Tool", which was connected to a regular NIC, both machines located inside a data center for testing purposes. The NICs were connected by a router and VLAN was set up to make it possible to use only MAC addresses for messaging between them. The idea was that PC-Tool would be used to generate a large enough packet stream to simulate the packet stream received by the actual industrial system and that Compute would receive the packets the NIC forwarded and measure the performance variables through a program. The processor used for processing outside the SNIC on the Compute-machine was an Intel(R) Xeon(R) Platinum 8160 CPU @ 2.10GHz. Intel's Data Plane Development Kit (DPDK) was presented as a way to provide for the traffic generating software and also contained API to create a program for measuring several network statistics.

Due to limited network access in the lab environment the machines were placed and two Docker images were set up to contain the necessary drivers for the SNIC (provided by Netronome) as well as the DPDK software. DPDK would prove to be necessary to set up a connection from within the hypervisors directly to the hosts' physical ports through its Poll-Mode Driver, eliminating any port-to-software delays. DPDK's pktgen was used as the traffic generator, and it was configured only to send IPv4 UDP GTPU packets.

After the images were set up containing the previously mentioned software, the image containing the statistics program and Netronome drivers were uploaded to Compute, and the image containing pktgen was uploaded to PC-tool. The firmware of the P4 program was generated using Netronome's programmer studio and the output of this was them loaded onto the Smart-NIC via Compute.

4.3 Performance Evaluation

The performance evaluation was solely based on packets per second. This was measured through a program using the DPDK API. The program was based on the DPDK provided example application "DPDK Layer 2 Forwarding Sample Application", which already contained functionality for measuring the amount of received packets. By adding code for measuring further statistics in the main program loop, a measurement for the packets received and processed per second was gained. Following is a detailed explanation on how the modifications to the program functions, followed by how testing was conducted, and a discussion regarding the possibility of integrated testing.

4.3.1 Measurement Program Implementation

Listing 4.1 shows the part of the program receiving the packets and executing the C-program for each one. The program begins by taking in packets in bursts of 32 packets and adds the amount of received packets to the port statistic struct on

```
1  nb_rx = rte_eth_rx_burst(portid, 0, pkts_burst, MAX_PKT_BURST);
2
3  port_statistics[1].rx += nb_rx;
4
5  for (j = 0; j < nb_rx; j++) {
6      before_processing = rte_rdtsc();
7
8      m = pkts_burst[j];
9
10     packet_length_t l3_length = (uint64_t)m->outer_l3_len;
11     packet_data_t*  l3_start  = (packet_data_t*)
12         rte_pktmbuf_mtod_offset(m, packet_data_t*, m->outer_l2_len);
13
14     if(!handle_classify_packet(l3_start, l3_length))
15         printf("Error, cannot execute C-program.");
16
17   after_processing = rte_rdtsc();
18   diff_processing  = diff_processing + after_processing - before_processing;
19 }
20
21 for (j = 0; j < nb_rx; j++) {
22   m = pkts_burst[j];
23   rte_prefetch0(rte_pktmbuf_mtod(m, void *));
24   l2fwd_simple_forward(m, portid);
25 }
```

Listing 4.1: Measuring program: packet count and C-program execution.

rows 1 to 3. Then the program starts looping through the packets, finding the length and start of the layer 3 protocol of each of them and uses these values to execute the C-program. Surrounding this processing are the `before_processing` and `after_processing` variables which uses the function `rte_rdtsc()` to measure the time in cycles since the program started. By then adding the difference between these to the variable `diff_variables`, a measurement of the time the total processing has taken can be gained.

When using the P4 classifier instead of the C-program classifier, a program extracting the data from the result header is executed. The final rows 21 to 24 are parts of the original Layer 2 Forwarding example-program, and sends the packets back to where they came from using another port.

Listing 4.2 shows the part of the program calculating the statistic measurements. Once again uses the function `rte_rdtsc()` is used for time measurement. The statistics are printed and calculated around each 10th and 40th second to avoid them being updated too quickly to read, hence the use of the `timer_period` variable, for measuring 10s intervals, as well as the `ten_second_loop_counter`, for measuring 40s intervals.

Note that depending on the time the latest receiving and processing loop took, the period may not always be exactly 10 seconds, but it was deemed that the deviations was small enough to not impact our results.

The program first checks so that the time of the period has passed and that the program is on the core handling the main port on rows 3 to 6. The `delta_pkt` variable is then calculated as the difference in packets received in total minus the

previously received packets from other iterations of the main loop. This value is then later used to calculate the `latest_rx_pps` on row 9 which is the packets received per second during the latest 10 seconds and is calculated as the `delta_pkt` divided by 10 seconds. `latest_receival_cycles_per_packet` is a variable used track the cycles each packet takes to be received and is calculated as the cycles during the 10 second period minus the cycles spent executing the C-program, divided by `delta_pkt`. The `latest_processing_cycles_per_packet` is the cycles each packet takes to be run through the C-program and is calculated using the processing time in cycles. The following variables with the same titles without beginning by "latest" are used to measure an average of the values over 40 seconds, which is done in the loop using modulus 4 on the current loop number to calculate the 40 seconds mark. After four 10 second loops, the average is calculate by simply dividing these variables by 4, creating an average. These averages are then summed to calculate the average total cycles per packet. Finally, this measurement is used to divide the cycles per second to get the total packets received and processed per second, which is the desired measurement value. Of course, variables are reset or set to appropriate values after each loop.

4.3.2 Testing Method

With both pktgen and the measurement program ready, the final resulting measurements were ready to be taken. The measurements for the SNIC were taken with the P4-program loaded onto the Smart-NIC and with the C-program execution commented-out. The processing time was then zero, and the measurements depended on the receival time.

The statistics for the C-program was taken with a P4-program accepting all packets loaded on the SmartNIC and the C-program execution un-commented.

The pktgen was started before the measurement program and the measurement program was run for circus 5 minutes (timed by hand). To check that the P4-programs was correctly loaded, GTPU packets was sent to check that they arrived or did not arrive respectively.

4.3.3 Possibility of Fully Integrated Testing

The original intention was to analyse the performance of the whole system with and without the SNIC integrated and compare these values. However, while this is possible, it was deemed to need the support of further personnel at the company, which probably would not have the time to offer their help on such quick notice. To avoid a deadlock situation, the partial integration described above was used instead. This way, one would be able to receive the necessary performance data in a less involved way.

```
cur_tsc = rte_rdtsc();

if (timer_period > 0) {
  timer_tsc += diff_tsc;
  if (unlikely(timer_tsc >= timer_period)) {
    if (lcore_id == 1) {
      delta_pkt= port_statistics[1].rx - prev_rx;

      port_statistics[1].latest_rx_pps                    = (delta_pkt)/10;
      port_statistics[1].latest_receival_cycles_per_pkt   = timer_tsc/(delta_pkt);
      port_statistics[1].latest_processing_cycles_per_pkt =
          diff_processing/(delta_pkt);

      rx_pps = rx_pps + (delta_pkt)/10;

      receival_cycles_per_pkt = receival_cycles_per_pkt +
          (timer_tsc-diff_processing)/(delta_pkt);

      processing_cycles_per_pkt = processing_cycles_per_pkt +
          diff_processing/(delta_pkt);

      if(ten_second_loop_no%4==0){
          port_statistics[1].avg_processing_cycles_per_pkt =
              processing_cycles_per_pkt/4;

        port_statistics[1].avg_receival_cycles_per_pkt = receival_cycles_per_pkt/4;
        port_statistics[1].avg_rx_pps = rx_pps/4;

        uint64_t tot_time_per_pkt = port_statistics[1].avg_receival_cycles_per_pkt
            + port_statistics[1].avg_processing_cycles_per_pkt;

        port_statistics[1].avg_handle_pps = (rte_get_timer_hz()*1.0)/(
            tot_time_per_pkt*1.0);

        processing_cycles_per_pkt = 0;
        receival_cycles_per_pkt   = 0;
        rx_pps                    = 0;
      }

      port_statistics[1].loop_number=ten_second_loop_no;

      print_stats();

      /* reset */
      ten_second_loop_no = ten_second_loop_no+1;
      timer_tsc          = 0;
      diff_processing    = 0;
      prev_rx        = port_statistics[1].rx;
    }
  }
  prev_tsc = cur_tsc;
}
```

Listing 4.2: Measuring program: statistics calculation.

4.4 Qualitative Evaluation Criteria

The qualitative evaluation consists of investigating the pros and cons of programming in the P4 language compared to programming in the C language, as well as weighing the limiting factors of using SNICs to the ones when using regular NICs.

For this qualitative evaluation, a collection of evaluation criteria has been set up. These are presented and briefly summarised in the list below.

1. Ease of environment setup, describing the difficulty of setting up an environment where the respective language can be compiled in.

2. Availability of documentation, describing the difficulty in getting information regarding the language, including functionality and modules.

3. Programming difficulty and limiting factors, describing the difficulty to first understand the syntax and how difficult it is to program when considering the language's limiting factors.

4. Hardware Limitations, investigating the limitations of using SNICs compared to NICs.

5

Results

This chapter presents the results obtained from the testing. The first section presents the results of the performance testing and the following section regards the qualitative testing, presenting the discovered challenges of using the SNIC and P4, including the difficulties of using the SNIC drivers as well as the limiting factors of P4.

5.1 Performance Evaluation

The testing, done as described in the previous chapter, focused on measuring the packet throughput in packets per seconds of the two setups when handling heavy streams of GTPU packets. The first setup being the P4 programmed SNIC with the classifier P4-program and the second the SNIC loaded with an empty program and the classifier C-program on the machine connected to it.

The result of the throughput measurement is shown in Figure 5.1 and indicates a 5 percent higher throughput for the SNIC focused setup, compared to the CPU focused setup.

When looking at the individual cycle measurements shown in Figure 5.2, it can be concluded that the processing time of running the whole classifier C-program on the CPU, need around 13 percent more cycles than when only picking out the necessary data from the result header after running the SNIC P4 classifier. From the same graph it can also be seen that the cycles for receiving the packets are the same in both setups, meaning the time it takes for the SNIC to run the classifier P4-program is negligible compared to the time it takes to run the empty P4-program.

Overall, the results point to the need to further optimize the way data is extracted from the result header to get better results.

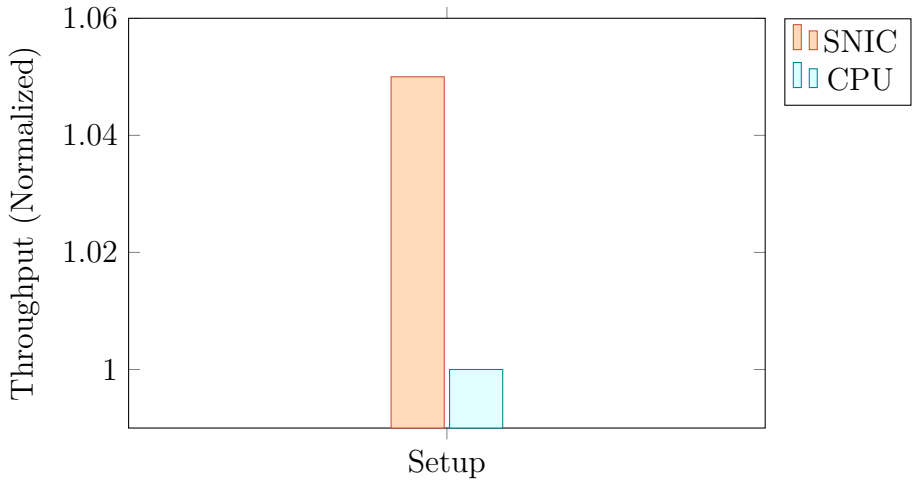

Figure 5.1: Difference in throughput between the SNIC implementation and the original system when handling GTPU packet stream.

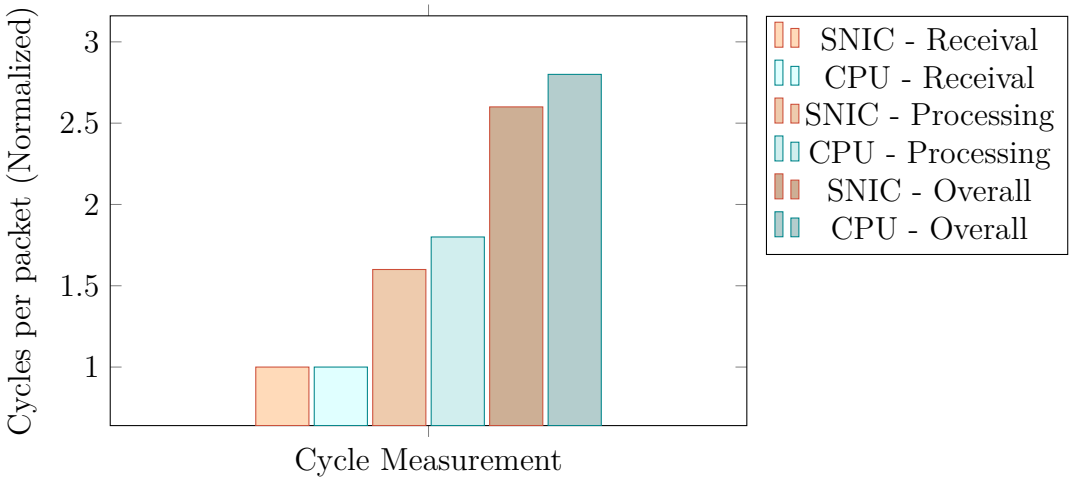

Figure 5.2: Difference in cycles between the SNIC implementation and the CPU system when handling GTPU packet stream.

5.2 Qualitative Evaluation

This Section describes the resulting evaluation of the qualitative evaluation criteria motivated in the previous chapter.

5.2.1 Ease of Environment Setup

During the first part of the project, the initial thought was to install P4 from scratch. However, after investigating further this seemed unnecessary complicated when the virtual machines already existed. Nevertheless, in the middle of the project, due to an issue that was thought to be due to the $P4_{16}$ compiler not being fully up to date (16.0 instead of 16.1) in the virtual machine, a few days were spent trying to install P4 from scratch on both a virtual machine using the same OS as the virtual machines (Ubuntu 14) and newer OS's (Ubuntu 16 and 20). The instructions used for the installation were the guides from the P4 repositories created by P4lang on GitHub. These contained a detailed guide on all dependencies needed, which were installed both manually and per a automatic script contained within one of the Github repositories. The conclusion of our efforts unfortunately proved pointless, as it seemed to always be some dependencies that could not be installed and it was unclear why. Proceeding without the dependencies also proved fruitless, since the installation then failed.

At this point, we deem it difficult to set up P4 from scratch, unless perhaps one has some experience with the needed dependencies, which we did not. However, when we further down the line decided to install the P4 compiler in one of the Docker images due to what was thought were problems with the compilation using Netronome's programmer studio, it proved to be surprisingly easy. We are not sure what was the problem with the previous installations, but we do recommend the using a hypervisor for anyone wanting to install the P4 environment on their own.

If not using Linux however, Netronome's Programmer Studio IDE for Windows needed no setup at all after installation, making it simple to start programming straight away. However, the Programmer Studio IDE has other problems, such as incomprehensible errors and possible bugs, as well as the lack of a working simulator for $P4_{16}$.

On the other hand, C is, due to its wide use, very simple to set up.

5.2.2 Documentation Availability

The documentation availability for $P4_{16}$ was limited, probably due to its novelty in the industry. The only up-to-date, extensive resource for information on the language was the P4 Language Specification ([39]) and the tutorials on P4Lang's Github and on the virtual machines. The P4 language website [1] further linked to the tutorials site open-nfp.org [51] where more tutorials were available, although the majority used $P4_{14}$.

Due to the lack of documentation, it was difficult knowing how exactly to use cer-

tain functionality in the language. These functionalities were usually seen used in arbitrary GitHub repositories and then one had to guess regarding whether or not they existed in the language as a concept or not. Registers were an example of such a concept. The use of registers were found in a repository and in the language specification but had to be implemented using different syntax than both of these sources to be able to be built. This makes one question if the language specification document also might be a bit out of date.

The limit in documentation also makes the different modules of the language somewhat mysterious. There is only brief texts on for example how the v1model and p4runtime work. The whole P4 language generally seems to be used within the group of people who are building it. However, over time this will hopefully change.

C is naturally very well documented.

5.2.3 Programming Experience

Although the lack of documentation is evident, the basic concepts of P4 are simple to understand after doing just a few of the tutorials on P4Lang's GitHub. The syntax is close to that of C and no extensive learning is needed.

As previously mentioned, there are several limiting factors of the P4 language, most prominent being the lack of loops and the non-variable sizes of header variables. The former is unavoidable when translating a regular program to pure P4. However, if the number of times the loop should be done is somewhat predictable, it is possible to unroll the loop to provide the same functionality. Of course, loops that are run an unpredictable amount of times are not translatable.

The non-variable size of header variables is also a limitation of the P4 language. This, however, seems to be a work in process where there is code being developed for measuring the size of a header, which pokes at the possibility that headers could be of different sizes. As of now, the header variables are set at defined sizes only, but perhaps this limitation will be fixed.

5.2.4 Hardware Limitations

Since the Smart-NIC basically is a regular NIC with added memory and computing power, there's no limiting factors of the Smart-NIC hardware that are not also limiting in the NIC. However, there are limits to the offloading that can be done with a SNIC when comparing offloaded and non-offloaded software. One of these is that the computing power that modern Smart-NICs contain are, in most cases, significantly less than those of the host processor. This means that the amount of offloading that can be done on the SNIC is limited to the processor hardware. The offloading capabilities are also limited to the amount of memory present on the SNIC.

5.2.5 Netronome

The P4 compiler issues as previously mentioned, proved not to be our only problem, since we would also need a backend compiler to be able to load the program to the specific Netronome hardware. For this purpose, Netronome provides a software development kit that can be used to both compile P4 programs and load the firmware and the configuration onto the SNIC. This software was not properly documented in Netronome's official guides, and nowhere was it mentioned whether it would be able to compile $P4_{16}$. Thinking it could only compile $P4_{14}$, some days were spent trying to find other solutions, when finally another thesis [52] was found to describe the certain command used to compile $P4_{16}$ to a specific Netronome Smart-NIC and load the compiled files onto the same.

Further, when attempting to load the program onto the NIC, using the method described in [52], additional problems appeared. While the method for compiling the program worked well, loading the files onto the SNIC did not. While it was possible to load the firmware using the guide described in [52] as well as the instruction found in the documentation for Netronome's SDK, the configuration caused unexpected errors. The configuration file in question was created using Netronome's Programmer Studio, and so syntactically correct, yet the software tool proves unable to load it onto the SNIC.

This issue was not resolved within the bounds of this project, and so we were forced to exchange the program for one that did not require a configuration file in order to perform any testing at all.

6

Concluding Remarks

In this chapter there will be a brief discussion of the results presented in Chapter 5 followed by a recommendation for future work and finally the conclusions reached by the authors during this project.

6.1 Discussion

This Section will contain a discussion of the results presented in Chapter 5.

6.1.1 Performance Evaluation

The performance evaluation conducted support the theory that performing packet processing on a SNIC rather than general use CPU is faster. The testing was however limited in scope, testing only GTPU streams and lacking the table configuration in the implementation. Because of this it is difficult to state with certainty that the results would not change if the scope was extended. Despite this, it is likely that the results of such testing would show a much greater improvement in comparative speed of the SNIC implementation. This is because the difference in processing time on the SNIC was negligible when performing the processing on the device compared to simply forwarding the packets. The implication of this is that the increase in processing time in the SNIC for extending the functionality is very likely to be much smaller than that of extending the C-program to perform additional tasks. Furthermore the time it takes the CPU to receive and extract the results from the P4-programmed SNIC would not increase despite adding logic to the program on the SNIC.

6.1.2 Qualitative Evaluation

The main problem that was encountered multiple times during the project was the lack of documentation. Further, many of documents that were found were outdated, such as the tutorials directed to by P4's site, which were mainly regarding $P4_{14}$, except 2 tutorials using $P4_{16}$. Another example was Programmer Studio which did not contain a usable version of the simulator, which also seemed to no longer be updated since the latest update from 2018. Many sources of information regarding $P4_{16}$ also seemed to no longer be updated. With $P4_{16}$ being quite new in terms of

programming languages together with the outdated material, it resulted in difficulty understanding the language on a deeper level, resulting in usage of parts of the language while they were still not fully understood.

Due to the amount of information on $P4_{14}$, as well as the working simulator, a question during the project was if it would not have been easier to use $P4_{14}$ instead. However, $P4_{16}$ is not backwards compatible with $P4_{14}$, which meant that there was a need to reprogram large parts of the program. The most severe problem was however the lack of a deparser in $P4_{14}$, which only emits the headers that came in in the same order, making it impossible to add a new header, making it necessary to rethink the whole result-struct approach. Also, since P4 does not operate on the payload of a packet, the only straight forward way would then be to change the existing headers to contain the result struct's values, which does not sound like a good solution.

6.2 Future Work

For future work the program should be tested as it was initially translated, with registers and tables. Without tables, there is no way P4 can be used in the way it is needed to be used in real scenarios, except perhaps very limited parts. It is therefore crucial to find out why the tables cannot be properly loaded onto the SNIC and fix it. Perhaps the best way of solving this is not using a Netronome Smart-NIC.

The dynamic table updates are also necessary to be examined for usage in the real environment. Therefore, the CLI need to be investigated more in detail to see that this can be done.

More measurements, perferably in a more simply controlled environment, should also be conducted for to gain a more precise insight in how much faster the Smart-NIC processing actually is compared to using a regular CPU.

Further, this project focused on a very specific part of the program at hand. An extension could for example be to translate the a classifier to P4.

Moreover, testing using different SNICs should be done and interfaces to update tables should be further researched to make sure they are available for all SNICs contrary to what Netronome has provided which may not be universally available.

6.3 Conclusion

The purpose of this project was to investigate how Smart Network Interface Controller can be used to offload packet processing in a realistic scenario, and identify the limitations and challenges of integrating the SNIC into an industrial system. The research that has been done in the years since the release of the language support the hypothesis that a P4 programmed SNIC can greatly increase the efficiency of packet processing compared to a general use CPU.

During the project, it was clear that one of the main limitations of the technology at this point is the lack of documentation for the product and the language. This made it difficult to with certainty determine the increase in performance and thereby the possible resources saved. As the finished test program was smaller than intended, the results are inconclusive. All tests performed do however still indicate that there is potential performance gain in using SNICs.

What can be said with some certainty based of the results of this project is that as of now, the Netronome SmartNIC, with at times lacking and/or incorrect documentation and an IDE with undefined behavior is not ready to be integrated in a real industrial system. There are simply too many unknown factors with the product to work with it as of now. Perhaps other SNICs are more documented and thus better candidates.

In contrast, the process has shown that the P4 language, is however ready to be used in real settings. It is a small and relatively easy language to understand that has good documentation and all parts of a basic C-program can be translated to. There's however more complicated behaviour that needs to be examined to know if all kinds of C-programs can be translated.

It is the belief of the authors of this thesis that with enough time, and perhaps further development of the products, using SNICs programmed with P4 to offload packet processing from general use CPUs could be key in facilitating the requirements of the 5G networks.

Bibliography

[1] Mihai Budiu and Chris Dodd. "The P416 Programming Language". In: 51.1 (Sept. 2017), pp. 5–14. ISSN: 0163-5980. DOI: 10.1145/3139645.3139648. URL: https://doi.org/10.1145/3139645.3139648.

[2] The P4 Language Consortium web site. 2020. URL: https://p4.org.

[3] *Cisco Annual Internet Report (2018–2023) White Paper*. Mar. 2020. URL: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html.

[4] M. Agiwal, A. Roy, and N. Saxena. "Next Generation 5G Wireless Networks: A Comprehensive Survey". In: *IEEE Communications Surveys Tutorials* 18.3 (2016), pp. 1617–1655. DOI: 10.1109/COMST.2016.2532458.

[5] Yanfang Le et al. "UNO: Uniflying Host and Smart NIC Offload for Flexible Packet Processing". In: *Proceedings of the 2017 Symposium on Cloud Computing*. New York, NY, USA: Association for Computing Machinery, 2017, pp. 506–519. ISBN: 9781450350280. URL: https://doi.org/10.1145/3127479.3132252.

[6] Z. Ni et al. "Advancing Network Function Virtualization Platforms with Programmable NICs". In: *2019 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*. 2019, pp. 1–6. DOI: 10.1109/LANMAN.2019.8847032.

[7] Salvatore Pontarelli et al. "FlowBlaze: Stateful Packet Processing in Hardware". In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 531–548. ISBN: 978-1-931971-49-2. URL: https://www.usenix.org/conference/nsdi19/presentation/pontarelli.

[8] Ming Liu et al. "Offloading Distributed Applications onto SmartNICs Using IPipe". In: *Proceedings of the ACM Special Interest Group on Data Communication*. SIGCOMM '19. Beijing, China: Association for Computing Machinery, 2019, pp. 318–333. ISBN: 9781450359566. DOI: 10.1145/3341302.3342079. URL: https://doi.org/10.1145/3341302.3342079.

[9] Y. Yan et al. "P4-enabled Smart NIC: Enabling Sliceable and Service-Driven Optical Data Centres". In: *Journal of Lightwave Technology* 38.9 (2020), pp. 2688–2694. DOI: 10.1109/JLT.2020.2966517.

[10] H. Harkous et al. "Towards Understanding the Performance of P4 Programmable Hardware". In: *2019 ACM/IEEE Symposium on Architectures for Networking*

and Communications Systems (ANCS). 2019, pp. 1–6. DOI: 10.1109/ANCS.2019.8901881.

[11] *Core Network Automation - Evolve your network*. 2021. URL: https://www.ericsson.com/en/core-network.

[12] 5G PPP Architecture Working Group. *View on 5G Architecture*. 2016.

[13] *Cloud Native Applications*. 2021. URL: https://www.ericsson.com/en/core-network.

[14] Irian Pupo, Alejandro Santoyo-Gonzalez, and Cristina Cervelló-Pastor. "A Framework for the Joint Placement of Edge Service Infrastructure and User Plane Functions for 5G". In: *Sensors* 19 (Sept. 2019). DOI: 10.3390/s19183975.

[15] J. Lewis and M. Fowler. *Microservices*. Mar. 2014. URL: https://martinfowler.com/articles/microservices.html.

[16] *Network Architecture Domains*. 2021. URL: https://www.ericsson.com/en/future-technologies/architecture/network-architecture-domains.

[17] SDxCentral Studios. *NFV Orchestrator (NFVO)? Definition*. Match 2016. URL: https://www.sdxcentral.com/networking/nfv/definitions/nfv-orchestrator-nfvo-definition/.

[18] *What if NFV*. 2021. URL: https://www.redhat.com/en/topics/virtualization/what-is-nfv.

[19] SDxCentral Studios. *Which is Better SDN or NFV*. Aug. 2013. URL: https://www.sdxcentral.com/networking/nfv/definitions/which-is-better-sdn-or-nfv/.

[20] *Network Exposure*. 2021. URL: https://www.ericsson.com/en/service-orchestration/network-exposure.

[21] *IP Multimedia Subsystem*. 2021. URL: https://en.wikipedia.org/wiki/IPMultimediaSubsystem.

[22] *What is the Evolved Packet Core and why should you care?* Sept. 2013. URL: https://www.telecomtv.com/content/news/what-is-the-evolved-packet-core-and-why-should-you-care-10610/.

[23] *User Plane*. 2021. URL: https://www.dialogic.com/glossary/user-plane.

[24] *Network Interface Controller*. 2021. URL: https://en.wikipedia.org/wiki/Network_interface_controller.

[25] *Network Interface Card : Types, Working, Advantages amp; Disadvantages*. Feb. 2020. URL: https://www.elprocus.com/network-interface-card-nic/.

[26] Daniel Firestone et al. "Azure Accelerated Networking: SmartNICs in the Public Cloud". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 51–66. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/nsdi18/presentation/firestone.

[27] *TCP Offload Engine*. 2021. URL: https://en.wikipedia.org/wiki/TCP_offload_engine.

[28] Jonathan Corbet. *Large receive offload*. Aug. 2007. URL: https://lwn.net/Articles/243949/.

[29] R. Bifulco and G. Rétvári. "A Survey on the Programmable Data Plane: Abstractions, Architectures, and Open Problems". In: *2018 IEEE 19th Interna-*

tional Conference on High Performance Switching and Routing (HPSR). 2018, pp. 1–7. DOI: 10.1109/HPSR.2018.8850761.

[30] *Intel® Smart Network Adapter (Intel® SNA) - SmartNICs from Intel*. URL: https://www.intel.com/content/www/us/en/products/network-io/smartnic.html.

[31] *SmartNIC - New Network Interface Controller*. Nov. 2020. URL: https://www.starwindsoftware.com/blog/smartnic-and-the-future-of-computing-storage-networking-and-security.

[32] Inc Netronome Systems. *Netronome NFP-4000 Flow Processor*. 2020. URL: https://www.netronome.com/media/documents/PB_NFP-4000-7-20.pdf.

[33] Scott Schweitzer. *SmartNIC Architectures: A Shift to Accelerators and Why FPGAs are Poised to Dominate*. July 2020. URL: https://www.electronicdesign.com/industrial-automation/article/21136402/xilinx-smartnic-architectures-a-shift-to-accelerators-and-why-fpgas-are-poised-to-dominate.

[34] Mellanox Technologies. *BlueField® SmartNIC for Ethernet*. 2019. URL: https://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf.

[35] Kevin Deierling. *What Is a SmartNIC*. Aug. 2018. URL: https://blog.mellanox.com/2018/08/defining-smartnic/.

[36] Inc Netronome Systems. *NFP-4000 Theory of Operation*. 2016. URL: https://www.netronome.com/static/app/img/products/silicon-solutions/WP_NFP4000_TOO.pdf.

[37] P. G. K. Patra et al. "Toward a Sweet Spot of Data Plane Programmability, Portability, and Performance: On the Scalability of Multi-Architecture P4 Pipelines". In: *IEEE Journal on Selected Areas in Communications* 36.12 (2018), pp. 2603–2611. DOI: 10.1109/JSAC.2018.2871288.

[38] Anirudh Sivaraman. *P4 language evolution*. https://p4.org/p4/p4-language-evolution.html. July 2015.

[39] The P4 Language Consortium. *$P4_{16}$ Language Specification*. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html#sec-packet-lookahead. May 2017.

[40] Inc Netronome Systems. *Programming NFP with P4 and C*. 2017. URL: https://www.netronome.com/media/documents/PB_NFP-4000-7-20.pdf.

[41] Palo Alto Networks. *GPRS Tunneling Protocol (GTP)*. 2021. URL: %5Curl%7Bhttps://docs.paloaltonetworks.com/service-providers/8-1/mobile-network-infrastructure-getting-started/gtp%7D.

[42] *Scapy's documentation*. 2021. URL: https://scapy.readthedocs.io.

[43] *Mininet*. 2021. URL: https://mininet.org.

[44] Inc Netronome Systems. *Datapath Programming Tools*. URL: https://www.netronome.com/products/datapath-programming-tools/.

[45] *Docker*. 2021. URL: www.docker.com.

[46] *DPDK*. 2021. URL: https://www.dpdk.org.

[47] Suneet Kumar Singh et al. "Offloading Virtual Evolved Packet Gateway User Plane Functions to a Programmable ASIC". In: *Proceedings of the 1st ACM CoNEXT Workshop on Emerging In-Network Computing Paradigms*. ENCP

'19. Orlando, FL, USA: Association for Computing Machinery, 2019, pp. 9–14. ISBN: 9781450370004. DOI: 10.1145/3359993.3366645. URL: https://doi.org/10.1145/3359993.3366645.

[48] Ericsson. *Evolved Packet Gateway*. 2021. URL: %5Curl%7Bhttps://www.ericsson.com/en/portfolio/digital-services/cloud-core/cloud-packet-core/evolved-packet-gateway%7D.

[49] C. Ed. Filsfils et al. *Segment Routing Architecture*. https://www.rfc-editor.org/info/rfc8402. 2018. DOI: 10.17487/RFC8402.

[50] Jacob Joubert. *P4 Introduction*. 2018.

[51] Open-NFP. *Open-NFP*. 2021. URL: https://open-nfp.org/.

[52] Jonatan Langlet. "Towards Machine Learning Inference in the Data Plane". In: (2019). URL: http://urn.kb.se/resolve?urn=urn:nbn:se:kau:diva-72875.