Offloading Virtual Network Functions – Hierarchical Approach

Jonatan Langlet

Faculty of Health, Science and Technology

Computer Science

Master’s Thesis, 30HP (ECTS)

Supervisor: Andreas Kassler

Examiner: Leonardo Martucci

Date: 150620

Preface

Dear reader,

The work presented in this thesis, *Offloading Virtual Network Functions - Hierarchical Approach*, was originally planned to be but one component in a much more ambitious project. A complete load balancing system was originally designed, which also included instructions running on a programmable switch. The final result would be a dynamic load balancing system performing 5G Core UPF functionality.

The goal was for multiple of these switches to work in parallel, each one responsible for user plane processing of a subset of total traffic flowing through a 5G core network. These switches could process some traffic themselves, and distribute the rest across multiple sub-processors. Due to issues with the control plane installation for the switch, and the thesis deadline approaching, I had no choice but to omit a lot of already completed work from this thesis, seeing as they would be of no use without a working control plane.

What *is* presented in this thesis however is the design, implementation, and performance of one of these network processor targets, planned to be deployed as a sub-processor under one of these switches. This target is a hybrid system consisting of DPDK applications running on an x86 server, and a programmable network card connected to the rest of the 5G core network. I want *you*, the reader, to keep this in mind when reading the rest of this thesis.

I hope you find the work interesting,

Jonatan Langlet

Abstract

Next generation mobile networks are designed to run in a virtualized environment, enabling rapid infrastructure deployment and high flexibility for coping with increasing traffic demands and new service requirements. Such network function virtualization imposes additional packet latencies and potential bottlenecks not present in legacy network equipment when run on dedicated hardware; such bottlenecks include *PCIe transfer delays*, *virtualization overhead*, and utilizing commodity server hardware which is *not optimized* for packet processing operations.

Through recent developments in P4 programmable networking devices, it is possible to implement complex packet processing pipelines directly in the network data plane; allowing critical traffic flows to be offloaded and flexibly hardware accelerated on new programmable packet processing hardware, prior to entering the virtualized environment.

In this thesis, we design and implement a novel hybrid NFV processing architecture which integrates programmable NICs and commodity server hardware, capable of offloading virtual network functions for specified traffic flows directly to the server network card; allowing these flows to completely bypass softwarization overhead, while less sensitive traffic process on the underlying host server.

An evaluation in a testbed with customized traffic generators show that accelerated flows have significantly lower jitter and latency, compared with flows processed on commodity server hardware. Our evaluation gives important insights into the designs of such hardware accelerated virtual network deployments, showing that hybrid network architectures are a viable solution for enabling infrastructure scalability without sacrificing critical flow performance.

Acknowledgements

First I would like to thank my advisor, *Prof. Andreas Kassler*, for his constant engagement and for introducing me to the world of computer networking research.
Thanks to the entire computer science department at Karlstad University, it has truly been a transformative time working with you.
Lastly, loving thanks to my partner *Mikaela* and our dog *Cisco*, for your never ending support.

Contents

List of Figures

List of Tables

Listings

1 Introduction

With Fifth Generation mobile networks (5G) on the horizon, Network Function Virtualization (NFV) has become an important tool that operators use to ensure that their infrastructure is able to scale alongside increased traffic demands [1]. In NFV environments, core network functionality is not delivered by legacy hardware, the virtualized network services are instead deployed as containers on commodity hardware.

Although migrating packet processing functionality to a NFV environment has clear benefits in terms of flexibility, it also comes with virtualization overhead and potential bottlenecks not present in legacy hardware; e.g. *network driver inefficiencies*[16], *PCIe bus transfers*[26], *software switch performance limitations*[20], and simply network functions executing on *unoptimized commodity hardware*[19].

A vision for future mobile networks is to have a broader impact than simply increased network speeds. One example is in health care, where some medical vehicles could be equipped with a remote-surgery station, allowing a surgeon to perform life-saving surgery immediately upon vehicle arrival [34].

These so called Ultra-Reliable and Low-Latency Communication (URLLC) scenarios require incredibly reliable mobile network connections, and a low perceived delay for the surgeon controlling the equipment remotely [15]. Allowing URLLC flows to be processed on more optimized systems than an NFV deployment is important for the flexibility of these new mobile networks, enabling accelerated processing of time-sensitive traffic.

One such Virtual Network Function (VNF), running in NFV environments, is GPRS Tunneling Protocol (GTP) encapsulation, which is a protocol allowing devices to move freely from one mobile network base station to another without interrupting ongoing connections for that device. GTP encapsulation is performed on every packet sent to the mobile device, resulting in increased traffic delays.

Recent developments in programmable network appliances make it possible to deploy complex functionality in hardware specialized at processing network traffic; one example of such programmable network equipment are SmartNIC (sNIC)s, which are P4 programmable Network Interface Card (NIC)s; P4 being a language designed solely for creating packet processing pipelines for deployment in network data plane equipment [4].

The objective of this thesis is to investigate the use of programmable network cards for accelerating time-sensitive traffic flows. This thesis aims to answer the question of how much packet processing latency can be reduced by offloading the packet processing to different programmable targets, and find the impact of processing complexity and lookup size on the acceleration targets.

By deploying VNF offloading P4 pipelines to the sNICs attached to these VNF hosting servers, one might assign certain time-sensitive traffic flows for processing directly on the sNIC; thereby allowing these critical traffic flows to completely bypass the overhead imposed by the NFV environment, while at the same time letting less critical traffic to continue down to the server for processing in the NFV environment as usual.

Equivalent P4 pipelines for processing GTP tunneling together with basic layer 3 firewall functionality has been designed and implemented on a programmable sNIC, and as a Data Plane Development Kit (DPDK) instance running on commodity server hardware used as a placeholder for a more complex NFV environment. The work presented in this thesis is a hybrid system with support for processing a subset of all mobile traffic flows on a server processor, while offloading other flows to the programmable sNIC. There are multiple benefits to this approach, including packets skipping the $900ns$ delay[26] imposed by its transfer through the PCIe to Central Processing Unit (CPU), and utilizing an architecture optimized for processing network traffic.

In this proof-of-concept implementation, flows that are processed directly on the sNIC measured approximately half as high processing latency compared to traffic processed on the equivalent DPDK pipeline running on host CPU; sNIC offloaded traffic also measured a significantly reduced latency jitter compared with non-offloaded traffic. Latencies for packets accelerated by the sNIC were in the $10us - 24us$ range depending on test scenario, while non-accelerated traffic processed on CPU measured in the $20us - 45us$ range.

The remainder of this thesis is structured as follows. Essential background knowledge is presented in section 2, starting with an overview of network function virtualization in 2.1, followed by the 5G Core Network (5GC) architecture in 2.2, GTP encapsulation in 2.3, and lastly P4 programmability in 2.4. How this work fits in the current state of research is explained in section 3. An explanation of the sNIC offloading solution is presented in section 4, where different offloading approaches are discussed. Final proof-of-concept implementation and test bed setup is described in section 5. Both individual processors, and the combined hybrid performance is presented in section 6, with a summary conclusion in section 7.

2 Background

2.1 Network Function Virtualization (NFV)

NFV is a concept that virtualizes entire network nodes, making these run as virtual machines - or containers - on commodity server hardware [6]. These nodes can be virtually connected together to deliver some specific network function. A VNF may run on one or more of these virtualized network nodes, delivering some network function without requiring specialized hardware designed specifically for this purpose.

NFV has reduced initial costs and simplified deployment of new network functions by running these on commodity server hardware [18]; virtualization has also improved the flexibility of deployments immensely [18].

There are however drawbacks to the NFV flexibility benefit though; NFV designs impose virtualization overhead, and is dependent on hardware which is not optimized for packet processing operations. Due to the non-deterministic behavior of x86 software stacks, they can not always guarantee strict latency requirements [28]. Multiple potential bottlenecks exist in NFV environments which are not present in legacy hardware, including *transmission through the PCIe bus* [26], *inefficient network drivers* [16], *software switch performance* [20], and the actual VNF executing on commodity servers which are *not optimized for these specific packet processing operations* [19].

Different network traffic types place their own requirements on the underlying network, e.g. video streaming does not necessarily require low latency, but do need a high data rate to ensure high quality streaming. However, for some kinds of augmented reality and remote surgery, ensuring a low packet latency is essential [15]; for these kinds of traffic, a hardware accelerated approach could be beneficial.

Even the most cutting edge virtualized environments struggle to deliver packet latencies as low as specialized hardware, as seen in a paper by Intel, where they manage a packet latency of $70\mu s$ for 5G User Plane Function (UPF) functionality running in a virtualized environment [9].

Thanks to new advances in P4 programmable data planes, some network devices can be extended to perform complex VNF functions themselves. In a paper by Singh et al, a high-end programmable switch can perform Virtual Evolved Packet Gateway (vEPG) functionality while delivering packet latencies as low as $2\mu s$ [31], thereby presenting a possible optimization for 5GC deployments.

2.2 5G Core Network Architecture

5GC is the next generation mobile network architecture, designed to replace the old Long-Term Evolution (LTE) *Evolved Packet Core (EPC)*. With 5G on the horizon, promising lowered packet latencies and higher throughput [32] compared to LTE, making it a very active field of research at the time of writing.

A central concept in the 5GC design is *Control and User Plane Separation (CUPS)* [29]. The control plane is responsible for signaling in a network, e.g. power management and attaching mobile devices, while the user plane is responsible for handling the actual user traffic passing trough the mobile network. As seen in figure 2.1, there is a very clear

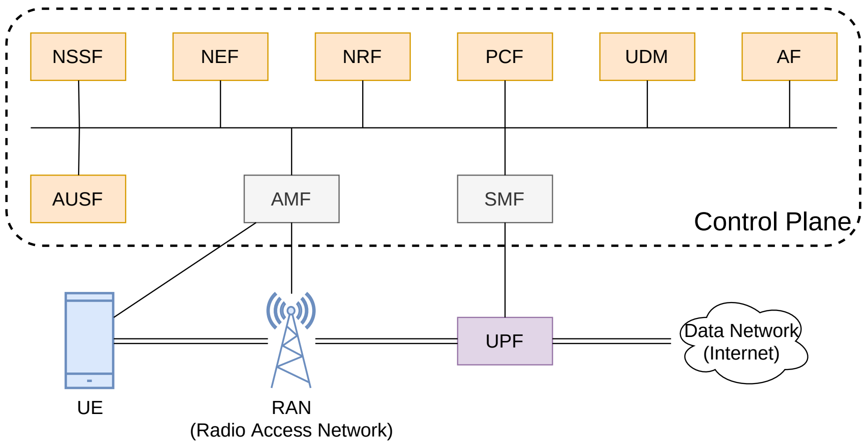


Figure 2.1: 5G Core Network Architecture overview - Showing separation of control- and user plane components

separation between the control- and user plane in the 5GC architecture. All control plane modules are also designed to be distinct, which means that they could easily be deployed as separate virtualized containers.

Since the 5GC architecture is designed for a virtualized environment [29], *network slicing* [11] is greatly simplified. Network slicing is a technique where multiple virtualized core networks can run in parallel; each slice is a complete core network with its own control- and user plane functionality. These slices can have their own network capabilities and services, and thus have their own specializations; some slices might be optimized for a high bit rate, sacrificing latencies to achieve this, while other slices aim to deliver low latencies and high reliability. One such traffic scenario is *URLLC* [12], which are traffic flows requiring low delays and high reliability, and is the case for certain augmented reality applications, remote surgery, robotics, etc. Mobile user equipment requiring these network properties can then be attached to a network slice which is optimized for processing URLLC flows.

To help operators implement network slicing technology, the 5GC architecture has been significantly revamped since its LTE predecessor. What in LTE is called *Serving Gateway (S-GW)* and *Packet Data Network Gateway (P-GW)* has in 5G been replaced

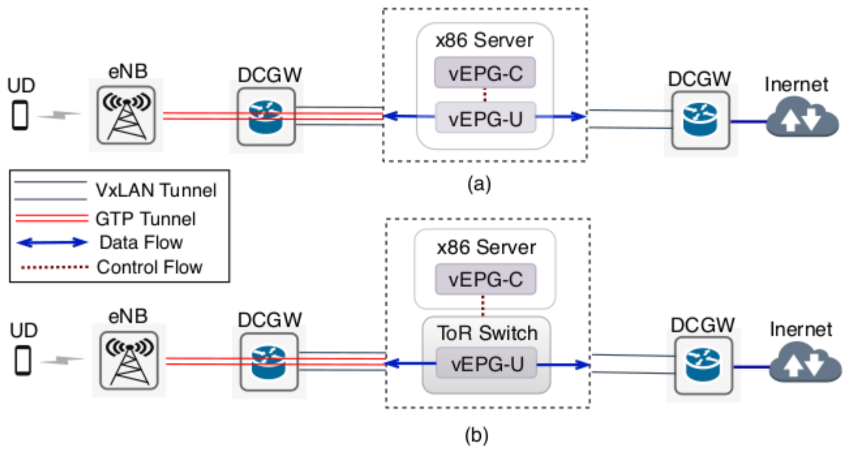


Figure 2.2: vEPG offloading explanation in LTE network, showing the GTP tunnel between RAN (eNB) and vEPG

by the UPF[7]. The UPF is responsible for facilitating user-plane functionality to the 5GC, including *access control*, *bearer lookup*, *service data flow (SDF) mapping*, *per-flow QoS*, *guaranteed bit rate*, *maximum bit rate (MBR)*, *charging*, *packet forwarding*, and *GTP encapsulation*[9]. Because of CUPS, and since all user plane functionality has been merged into a single user plane module, the UPF is also designed for a virtualized environment[30]. When the user plane load increases in the packet core, more UPF VNFs can immediately be deployed to offload the current system, thereby allowing the 5GC to scale alongside increased network traffic demands.

2.3 GPRS Tunneling Protocol (GTP)

An example of a network function which might benefit from hardware acceleration is *GTP encapsulation* in the 5GC, since these operations are affected by the NFV bottlenecks as explained earlier in section 2.1.

vEPG is a variant of Evolved Packet Gateway (EPG), which is performing various network functions in the EPC, including GTP user plane processing and basic firewall functionality. A paper by Singh et al. [31] evaluated the benefit of offloading vEPG

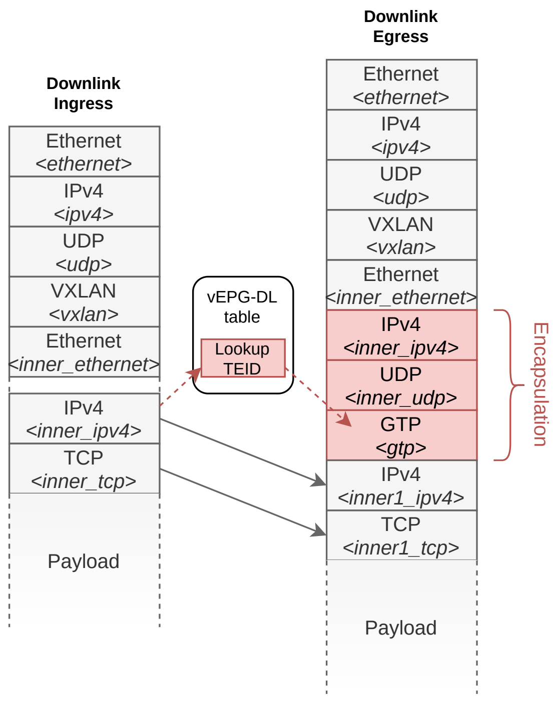


Figure 2.3: GTP encapsulation process in EPG

functionality to a programmable packet switching Application Specific Integrated Circuit (ASIC), more specifically a Barefoot Tofino.

Singh's paper presents a decent visualization of where the GTP tunneling occurs in the LTE *EPC*; see figure 2.2, where the *Data Center Gateway (DCGW)* on the left is to the *Radio Access Network (RAN)* where the E-UTRAN Node B (eNodeB) will send the packet to the mobile device, and towards the Internet on the right.

In (a), both the vEPG controller- and data plane instances are running as VNFs on commodity x86 server hardware, and a server CPU is responsible for all network processing. In (b) however, the whole data plane instance has been offloaded into the Tofino *Top-of-Rack (ToR)* switch, leaving just the controller VNF on server CPU.

GTP tunneling is responsible for ensuring packets coming from the Internet are routed to the correct mobile base station, to which the destination mobile device is currently attached. GTP tunneling allows for mobile devices to move from one base station to

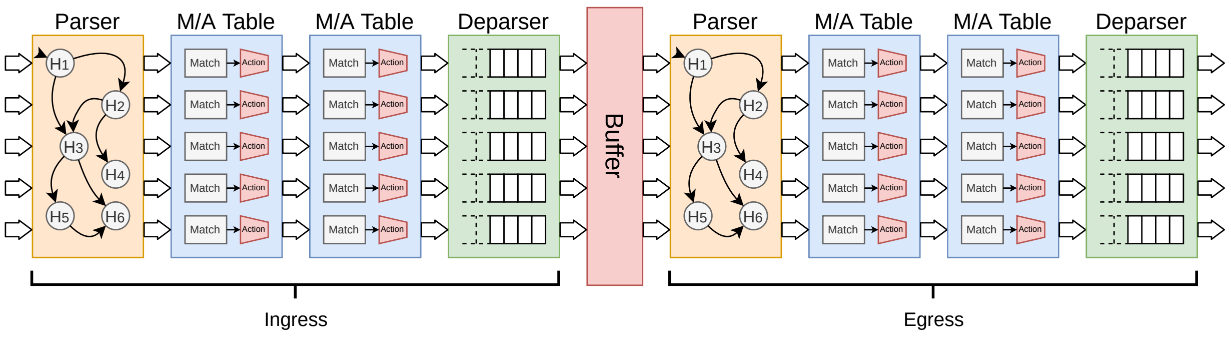


Figure 2.4: Overview of a P4 packet processing pipeline. Different P4 targets can have their own variations on this pipeline structure, but always contain these same basic components

another, without breaking currently ongoing internet connections. To achieve this, a *Tunnel Endpoint Identifier (TEID)* is inserted in the GTP header, and kept updated by the core network controller; these TEIDs are unique integers used to identify a tunnel between endpoints as shown in figure 2.3. Seeing as there can be millions of user tunnels in the same core network, this requires extensive lookup tables to store all IP/TEID mappings.

In modern mobile packet core networks, GTP encapsulation and firewall functionality is performed on commodity x86 servers in an NFV environment, as is the case with vEPG [31].

2.4 P4 Programming Language

P4 is a programming language designed exclusively for data plane programming. The idea was first published as a SIGCOMM paper in 2014 [4], and has since then grown into the de-facto standard language used for data plane programming.

2.4.1 P4 Design Philosophy

The P4 programming language is used to describe real-time packet processing pipelines targeting very diverse hardware, including *Field-Programmable Gate Array (FPGA)s* [21], *commodity servers* [33], *NICs* [22], and *ASICs* [25]; a lot of standard components typically included in imperative programming languages are therefore omitted from P4 - including *all loops*, *division and modulo operations*, *floating-point calculations*, *dynamic memory allocation*, *recursion*, etc.

Packets processed by a P4 program always enter at the same place, namely the *Parser*. The parser is a fully programmable *finite state machine*, responsible for parsing the packet headers. In each parser state, a header can be extracted; state transitions can be chosen based on header values parsed earlier, and can therefore ensure correct parsing of subsequent headers in the packet. For example, the *Ethernet header* can be the first parsed header, and based on the *EtherType* field, the parser can transition to a state responsible for parsing the Internet Protocol (IP) header, Address Resolution Protocol (ARP) header, etc. Only header values extracted during this parser are available for use further down in the P4 pipeline.

When parsing is finished, packets enter a match-action part of the *Ingress block*, where most of the processing logic occurs. The main idea of P4 match-action processing is to execute specific actions based on entries in lookup tables. These match-action tables are populated by the control plane with key values with which packets will attempt to match against. Upon successful match against a table entry, a specified action is triggered along with input parameters specific for the rule which was matched against. These actions typically work like functions in traditional imperative programming languages, and can be thought of as such.

After the match-action pipeline is applied, the packet reaches the *deparser*. In here, headers and payload are stitched back together in a pre-defined order ready for emission. The ingress pipeline is followed by an egress pipeline, which follows the same design philosophy as the ingress pipeline. The ingress- and egress pipelines are often programmed to perform different sets of instructions, and the ingress pipeline can even instruct certain packets to completely bypass the egress pipeline. This can be useful in cases where only some packets require further processing, which then will be placed in the egress block.

Different P4 programmable hardware can have their own variations of this basic P4 pipeline structure, where some targets might place an extra deparser and parser stage in between ingress and egress as in figure 2.4; others might go directly from the ingress match-action pipeline to egress, barely distinguishing between the two.

P4 does not support any complex data types. Instead, developers explicitly specify the bit-length, and can use basic integer and bitwise operations for processing the stored values.

There are a few options for data storage in P4 programs, including the ones mentioned below.

Metadata is very similar to local variables in traditional languages. These are used to store temporary values during a single packet processing, and can not be used for persistent storage.

Registers are used to store data persistently across packets, and are therefore useful while developing stateful P4 programs. Registers are arrays, and the developer must specify how many register instances should be allocated during compile-time. These registers are not 100% standardized in P4. P4 target architectures are diverse, and it is difficult to define a method of persistent storage which is guaranteed to work on all hardware. For example, some hardware targets place limitations on how often and in what way these registers can be accessed.

Counters are bound to tables, and increment every time a specified action is triggered. These are simple persistent storage components, and can be useful in cases where the program is dependent on the knowledge of how many times an action has occurred.

Meters are complex components used for keeping statistics, often related to packet- or bit rates. These components can return either *green*, *yellow*, or *red* depending on the rate with which they have been triggered during some interval; the thresholds required meters to report any specific color is specified by the controller.

2.4.2 P4 Compatible Hardware

Even though the P4 language is still in it's early stages, there are already multiple producers of P4 programmable network hardware.

Barefoot Networks is perhaps the biggest actor in this field, being the creators of the P4 programmable packet switching ASIC called *Tofino*[25]. Tofino is an incredibly powerful P4 compatible switch, able to process network traffic at speeds up to *6.5Tbps*.

Another big producer of P4 programmable hardware is Netronome, who focus on developing sNICs [22]; which are relatively inexpensive P4 programmable NICs.

2.4.3 Netronome SmartNIC

Typically, NICs are fixed-function devices hard-wired to perform a simple forwarding function, providing an interface between a computer and an external network. Netronome Agilio CX is a series of fully programmable devices, developed by Netronome[24], built around their *Network Flow Processor (NFP) architecture.*

A high level of parallel processing is at the base of this hardware architecture, designed to achieve a high packet processing performance[23]. This flow processor architecture is based around multiple processing cores, called workers. There are 60 flow processing workers, each one clocked at $633MHz$ shared by eight threads; all threads are executing the same packet processing program. These workers are clustered together into multiple worker islands specialized for certain packet processing operations.

Worker islands all have their own sets of memory regions, and this architecture uses a spillover technique while allocating memory, i.e. the fastest memory regions are prioritized while allocating memory, and when these fill up, new allocations are 'spilled over' into slower, but larger, regions.

Netronome sNICs are programmable in a language called Micro-C, which is a stripped down version of C designed to run in a real-time environment with additional limitations not present in a traditional CPU environment. Micro-C is completely lacking in dynamic memory, requiring all memory allocations to be known at compile-time, and due to the lack of stack memory does not support recursion. On top of this, there is also no inherent support of floating point calculations, and all but the most basic C libraries are not included.

Netronome does provide a P4 compiler, which converts a P4 program into an equivalent Micro-C code. The generated Micro-C code is then compiled into the firmware binaries loaded onto the sNIC device. Because the building process is done step-by-step, it is possible to modify the generated Micro-C code before its compilation if very low-level modifications to the sNIC firmware is desired. An example of such a modification is briefly discussed in section 6.2.

Through Single-Root Input/Output Virtualization (SR-IOV), it is possible for user space applications running on the underlying host to directly receive network traffic from the network card, completely bypassing the inefficient kernel space. When a NFP sNIC is installed in a motherboard with SR-IOV support, the P4 pipeline running on the sNIC can perform packet processing and afterwards forward packets through Virtual Function (VF) ports down to host, where a user space application is ready to receive it.

2.4.4 T4P4S and DPDK

DPDK is a set of data plane libraries and drivers which enables advanced data plane functionality to be performed on traditional Linux servers [14]; such data plane functionality includes GTP processing nodes, packet filtering firewalls, and virtual network switches.

Network traffic coming through the NIC is usually handled by *kernel space*, which is an inefficient interrupt-driven environment; each packet processed by the kernel space would throw an interrupt, telling the CPU to perform a context switch and immediately process this incoming request.

What DPDK does instead is allowing incoming network packets to completely bypass the kernel space, through the use of a *Poll Mode Driver (PMD)* which access the Receive

(RX) and Transmit (TX)-queues of the NIC directly; resulting in greatly improved packet processing performance, by keeping all packet processing in the user space without imposed kernel overhead.

Traffic coming from the NIC is distributed through a *Receive-Side Scaling (RSS) hash function* to one of the RX-queues. Each one of these RX-queues only support a single core to poll packets; each core can be instructed to poll packet from any number of queues, as long as it does not share an RX-queue with another core. Because of this, the number of supported RX-queues places a limitation on the maximum number of cores DPDK can use for processing network traffic.

T4P4S is a P4 compiler for DPDK, making it possible to run P4 applications in a DPDK environment [33]. Setting up a T4P4S application requires both P4 code for compilation, and a set of configuration options - which includes specifying numbers of RX-/TX-queues, and number of cores together with a mapping between these queues and cores as described earlier.

T4P4S applications also require matching control plane instances, responsible for control plane functionality. Controllers in T4P4S handle rule insertions and updates in match-action tables, modifying register values, configuring meters, etc. These control plane applications are written in C, made specifically for the T4P4S application it is responsible for.

3 Motivation

In a multi-year long study by Lévai et al., published in 2018, they dive into the price of including programmability in the software data plane [17]. Traditionally, networks are entirely built around fixed-function devices, where the functionality is hard-wired into the hardware. This has resulted in very good processing performance, but greatly limits what can be done in the network data plane. For example, they argue that a pure fixed-function network makes it very difficult to support the next-generation 5G mobile infrastructure. Although, enabling more programmability in the data plane comes at a cost both in terms of forwarding performance and hardware expenses. Lévai et al. came to the conclusion that current programmable networks struggle to scale to the same load as traditional fixed-function hardware devices, and call for further research into scalable programmable networks.

In 2018, Neugebauer et al. measured the PCIe impact on network performance [26]. During their tests, they found an imposed additional packet latency of $900ns$ for passing 128B packets through the PCIe down to the host. Offloading processing of these packets to the NIC would entirely eliminate the need for this PCIe transfer, and thereby reducing the overall latency for offloaded traffic.

Fortunately, allocating traffic flows to different processors, according to the specific needs of each flow, has been a topic of research for multiple years. Abdelwahab et al. published a paper in 2016 where they discussed the idea of deploying highly deterministic and specialized hardware at the network edge, responsible for handling flows requiring these specific network properties [1]. A system capable of processing some traffic flows before even reaching the NFV environment could be used to improve the network performance for these time sensitive communications.

In an effort to improve on the issues presented above, this thesis will focus on enhancing the scalability of programmable data planes, and deployment of efficient VNFs, by offloading these operations from servers onto programmable NICs. These networks cards are designed with a high level of parallelization, which is lacking in traditional x86 architectures. Currently, VNF operations are performed in virtualized containers on traditional x86 servers, with network packets passing through a traditional non-programmable NIC. If these NICs are made programmable, a subset of network traffic could be offloaded directly to the NIC without having to pass by the underlying host server.

In this thesis, packet processing pipelines have been developed and deployed into a system as described above. This thesis is evaluating the performance gain for offloading GTP encapsulation together with layer 3 firewall functionality, which are both functionalities included in modern mobile packet core networks, onto programmable NICs.

4 Design

A hybrid packet processing system pipeline has been designed, which enables offloading of GTP encapsulation and basic layer 3 firewall functionality onto a programmable sNIC, as seen in figure 4.1.

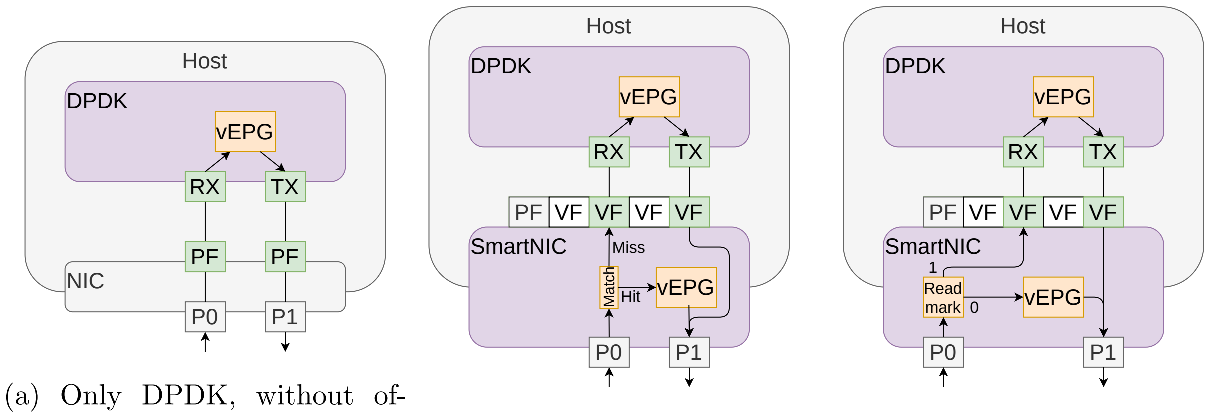
Equivalent pipelines have been created for an x86 packet processor, and a programmable sNIC attached to that packet processing server. The NFP architecture on the sNIC is based on a higher level of parallelization than the server, with 60x633MHz packet processing workers compared to the 10x2.2GHz CPU cores running on the host; it might therefore be reasonable to assume that the sNIC, optimized for the parallelizable nature of network packet processing, could deliver reduced latency jitter compared to the CPU.

The following sections will describe the hybrid pipeline, designed to offload a subset of total network traffic for sNIC acceleration; thereby attempting to reduce latency and jitter for these traffic flows.

4.1 Offloading Overview

There are numerous design possibilities for offloading VNFs, and this section will focus on offloading vEPG functionality as briefly explained earlier in section 2.3.

In figure 4.1, three different solutions are presented. The left-most scenario, without offloading, is explained in section 4.1.1; section 4.1.2 explains the middle approach, where a simple sNIC pipeline could be used to offload the DPDK; while section 4.1.3 explains the right-most offloading approach, made possible by coordinating with a traffic load balancer.



(a) Only DPDK, without offloading

(b) DPDK, offloaded by SmartNIC pipeline

(c) DPDK, offloaded by enhanced SmartNIC pipeline

Figure 4.1: Different ways of performing vEPG functionality, with and without SmartNIC offloading

4.1.1 DPDK Without SmartNIC Offloading

DPDK instances are typically bound behind traditional fixed-function NICs, which are not programmable. In these cases, there are no possibilities to use the NIC for offloading complex network operations. Instead, the NIC is simply an interface between a computer's internal software, and traffic on the network medium. In this scenario, DPDK is bound directly to the Physical Function (PF) ports of the NIC, and has full responsibility for processing all traffic sent to that specific NIC. Figure 4.1a shows a visualization of this scenario.

4.1.2 Simple SmartNIC Offloading

Thanks to P4 programmable NICs, called *sNICs*, a relatively simple pipeline can be deployed on the NIC to offload DPDK, and therefore reducing load on the host system.

If DPDK is bound to SR-IOV interfaces for the sNIC, the P4 pipelines running on these two devices could be made to work together. Using GTP encapsulation as an example, the total number of mapped TEIDs could be split so that each target is only responsible for a subset of total encapsulations. The sNIC pipeline can be designed with an equivalent encapsulation table to the DPDK, and only forward packets to DPDK if the tunnel is not found in the sNIC table instance.

When a packet enters the sNIC, a table lookup is first performed in the sNIC instance of the GTP encapsulation table. If there is a miss, the packet is forwarded through SR-IOV to the host, where DPDK is bound to the VF ports and continues processing of this packet. Assuming the tables in DPDK and sNIC are populated so that their union is equal to the full set of TEIDs, this packet will successfully find an entry in the DPDK table, and GTP encapsulation will be performed by the DPDK process. However, if the control plane had created an entry for the packet in the sNIC table instance, encapsulation would have been performed directly in the NIC, and thereby completely bypassing the host machine and its inefficiencies as explained in section 2.1. This approach is visualized in figure 4.1b

4.1.3 Offloading With Marked Fast Path to DPDK

For the packets which only have a table entry in DPDK, attempting to match against the GTP table in the sNIC is a suboptimal solution, resulting in unnecessary table lookups. If it is assumed that some load balancing device has already forwarded the packet to this specific destination, one might also expect this load balancer to be capable of marking the chosen destination into the packet during load balancing.

Figure 4.1c shows how this marker could be used to bypass the sNIC table, instead using the parsed marker to beforehand know which target will perform the encapsulation.

If a table entry for the packet currently in the pipeline has not been installed in the sNIC table instance, attempting to match against this entry would result in a table miss. If instead this marker already explicitly mentions a different target than the sNIC, the table could be completely bypassed. Instead, the parsed marked value can be used to forward the packet down the hierarchy to the correct encapsulation device, which would be DPDK in this case. However, if this marker specifies the sNIC as encapsulation target, an entry for this packet should have been written to the table, and a lookup is performed directly in the sNIC pipeline.

These packets could be marked using a *Network Service Header (NSH)* [27], which is used to ensure that packets are routed through a specified sequence of service nodes. In this case, the vEPG processing target would be encoded in NSH, and if the target parses its own ID in this header, it knows that an entry for this packet should be present in the encapsulation table.

Due to this work only being a proof-of-concept implementation, the marker is instead encoded into the *IPv4 Identification field*, containing a 16-bit value used for IP fragmentation which never occurs during these experiments. The goal here is to evaluate the performance impact of a marker which explicitly specifies its vEPG processing target to be used for bypassing unnecessary table lookups, and encoding this marker here works as a proof of concept.

Having the target marked in the packet also enables more complex hybrid system designs. In the design presented in this thesis, packets which are not processed in the sNIC are always sent through the same VF-port where DPDK is bound; a marker design could instead specify which exact host service should continue processing, by having specific packet processing services bound to their own sNIC VF-ports.

4.2 vEPG Pipeline

All targets are executing equivalent mobile packet core UPF pipelines, focused on processing downlink traffic coming from data center gateway router on its way to the correct base station where the mobile device is attached; supporting GTP encapsulation and decapsulation, together with simple layer 3 firewall functionality. This firewall can be instructed to drop packets based on exact-match of the *inner_ip header* destination address. These two combined functionalities, i.e. layer 3 firewall and GTP processing, will here be referred to as vEPG functionality.

```
if(  standard_metadata.ingress_port == 0 ) //If this packet is from traffic generator
{
  smac.apply();
  dmac.apply();
  if( hdr.ipv4.isValid() )
    if (hdr.gtp.isValid())
    {
      firewall_UL.apply();
      vEPG_UL.apply(); //GTP decapsulation.
    }
    else
    {
      firewall_DL.apply();
      vEPG_DL.apply(); //GTP encapsulation. Default: send to host. On hit: send to TG
    }
}
else //If pkt from not from TG, assume from host
  send_to_tg(); //Forward packet to traffic generator
```

Listing 1: Snippet from SmartNIC ingress pipeline, offloading without marker

4.2.1 Parser

As mentioned previously in section 2.4, all packets start their processing in the packet parser, where header values are extracted for use further down in the processing pipeline. Due to the complex structures of packets passing through the packet core, the parser requires a relatively high complexity to support parsing of these packets. Figure 4.2 visualizes the parser as designed and implemented for these tests.

4.2.2 Ingress

In this implementation, most of the functionality is placed in the ingress pipeline. A code snippet from the ingress pipeline running on the offloading sNIC without marker handling is presented in listing 1, used for the offloading approach mentioned in section 4.1.2.

Layer 2 tables are the first to be applied, which are the *smac* and *dmac* tables; smac can be used for Medium Access Control (MAC) learning, and dmac is used to whitelist destination MAC addresses. For these tests, MAC learning is not required since the for-

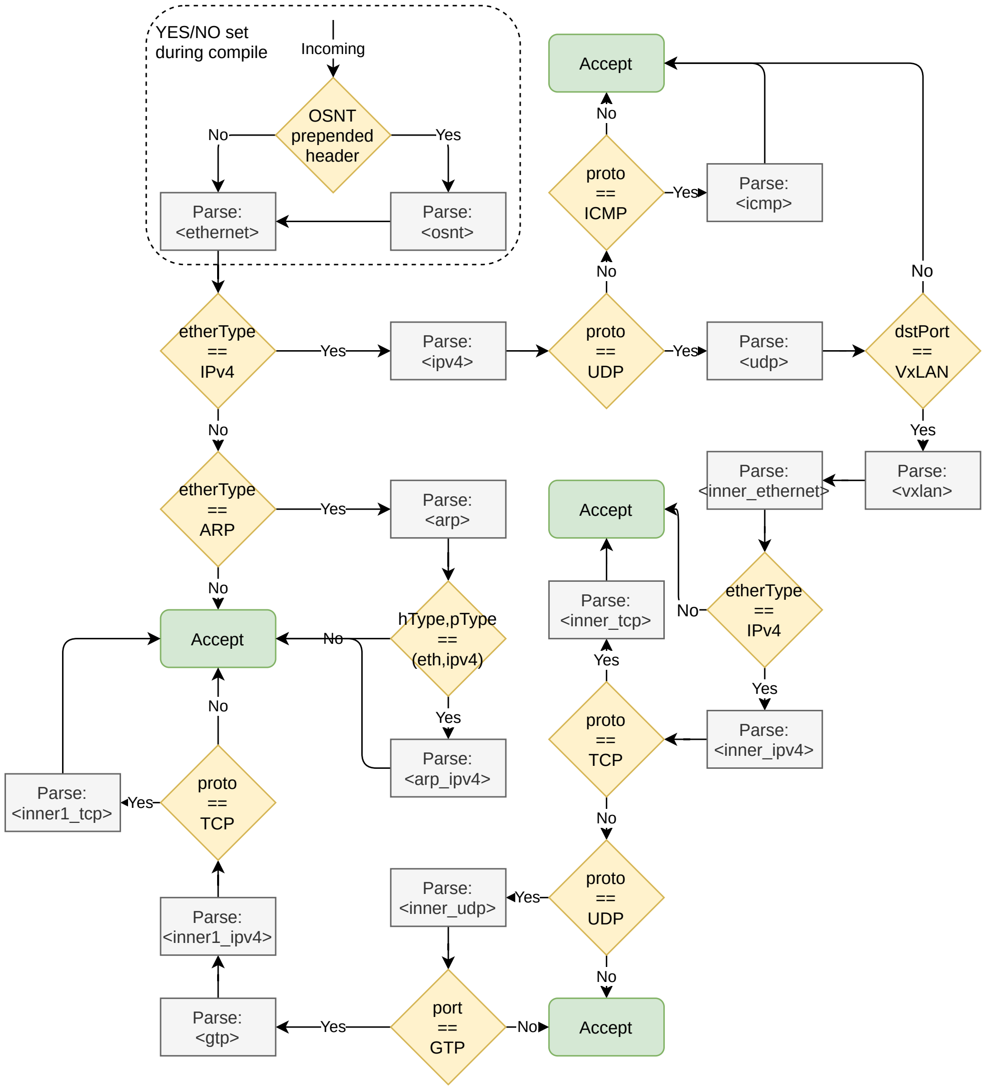


Figure 4.2: A visualization of the vEPG P4 pipeline packet parser, as it was designed for this project. This state machine is capable of parsing the complex header structure of packets in the mobile packet core

warding rules are hard coded as shown in figure 4.1, and the MAC learning action triggered from the smac table is therefore not filling any function. To keep the pipeline functionally close to an actual packet core UPF, both layer 2 tables are still present.

For layer 3 processing to be performed at all, a single conditional verifies that the *ipv4 header* was parsed. Packets are processed by the uplink branches, i.e. deparsed, if the *GTP header* was parsed; if a GTP header was not parsed they are processed as downlink packets, and are encapsulated. Both the uplink- and downlink branches contain layer 3 firewalls, which can instruct the device to drop traffic based on addresses in the *inner_ip header*.

4.2.3 Egress

In this GTP processing implementation, no functionality had to be placed in the egress pipeline. Both targets (T4P4S and sNIC) share resources across ingress and egress, which means that leaving this block empty should not have a noticeable impact on the system performance.

Load balancing functionality could be placed in the sNIC egress block for traffic distribution across host processes, as explained in section 5.3.1; which would include calculating a hash based on parsed packet header values, and mapping the resulting hash to specific virtual ports towards the host. However, the design implemented here does not include this functionality.

4.2.4 Deparser

As mentioned earlier, the deparser is responsible for emitting headers in a pre-defined order, will only emit headers which are set as valid during either the parser or ingress pipeline.

4.2.5 Encapsulation Action, and Optimization

During the later part of the project, an optimized encapsulation action was designed. The old encapsulation action worked as follows:

1. Copying inner_TCP and inner_IP header values into metadata
2. Setting inner_TCP header invalid, preventing header from emitting in deparser
3. Setting inner_UDP, GTP, inner1_IP, and inner1_TCP headers as valid, making them emit in deparser
4. Populating inner_TCP and inner_IP headers with values stored in metadata
5. Populating GTP header using hard-coded constants, and the TEID retrieved from match-action table
6. Updating ethernet, IP, UDP, inner_ethernet, inner_IP, and inner_UDP using constants

The optimized encapsulation action completely bypasses metadata by instead copying inner_TCP and inner_IP directly into the headers below GTP encapsulation. This optimized solution was then ported to both targets, since both have been confirmed to support direct cloning of entire parsed headers.

4.2.6 SmartNIC Offloading Functionality

In the Netronome sNIC ingress pipeline, there is a small addition to enable forwarding of some packets to the host CPU. There are two design approaches evaluated in this thesis, which are both explained previously in section 4.1.

The simple approach described in section 4.1.2 is implemented by modifying the sNIC GTP encapsulation table, making the default action in that table forward packets to the host.

In the marker offloading approach from section 4.1.3, a simple conditional statement is included in the ingress pipeline, comparing the parsed marker value against a constant defined while compiling firmware for the targets. The sNIC downlink branch is only applied if this parsed marked value is equal to the constant, as defined at compile-time and encoded in certain packets where sNIC acceleration shall be performed. Only packets with a marker value not matching the sNIC constant will continue down to the host for non-accelerated processing.

5 Implementation and Methodology

This project has been predominantly practical in nature, requiring installation, configuration, and management of multiple software as well as hardware systems. Due to this, setting up the test environment has without question been the most time consuming part of the project. [1]

5.1 Generating Test Cases

For testing the performance of the hybrid target, packet traces together with matching table entries had to be generated for each test scenario. To ensure consistency across tests - including matching traces and table rule configuration files - a metadata file called *flows.json* has been generated. The flows-file has been used to store a list of one million IP addresses, together with their corresponding TEID values; each JSON entry contains source and destination IP addresses and an integer value ranging from 0 to $2^{31} - 1$. All IP addresses and TEIDs used are unique, meaning each value only appears at most once in the flow-file, which is only generated once at the beginning of the project, and the values stored within are used as base while generating traces and configurations for all tests. Each

[1]As described in the thesis preface, I also installed and configured a Barefoot Tofino switch, which by itself required almost a month of my time. Unfortunately, there were issues with the control plane interface for the switch, which is the reason why it is not included in the final thesis version

```
{"srcip": "2.144.111.207", "dstip": "205.212.147.22", "teid": 644680905},
{"srcip": "182.199.190.59", "dstip": "37.181.194.235", "teid": 604547847},
{"srcip": "228.220.128.225", "dstip": "88.139.83.242", "teid": 765561505},
```

Listing 2: Snippet from flows.json file, used to ensure flow consistency across tests. All generated traffic flows and table rules are based on the values in this file

time files are generated for a new test case, *flows.json* is read into a global array called *iptable*, which is referred to each time these values are needed.

Listing 2 present a snippet from the flows.json, showing the first few flow entries. Since these are the first three entries in *flows.json*, these are also the addresses which will be encountered in the first three packets during all tests; where these addresses can be found inside the *inner_ipv4* as shown earlier in figure 2.3, and the match-actions tables will be configured to encapsulate these packets with the TEIDs as shown in the snippet.

5.1.1 Generating Packet Traces

Packet traces are generated in a Packet Capture (PCAP) format using *Scapy*. These files contain pre-generated network traffic, where raw packets are stored in the exact same format in which they will be replayed through the equipment. The packet generation function can be seen in appendix A.2.1, and takes 7 input parameters which defines the generated trace as follows:

count How many packets that should be generated and included in this trace. These packets are generated according to entries in *flows.json*, sequentially starting at the beginning, so that a single packet is included for each of the first *count* entries in the flows-file.

psize The size of packets generated and included in the trace. These traces only contain packets of equal size; total size is including size of headers, and a random payload is appended to the end of the packet so that total size of each packet equals *psize*.

uplink This specifies if the trace should contain uplink- or downlink packets, and will then only contain packets of this type. Uplink packets are already GTP encapsulated, while downlink packets are not GTP encapsulated. The TEID for uplink packets is read from *flows.json* to match the IP addresses.

tsprepend When this is set to *True*, a 128bit Open Source Network Tester (OSNT) header will be prepended to each packet, as explained later in section 5.4.1. This header would if so contain three elements: a 32-bit signature, a 32-bit packet counter, and a 64-bit transmit timestamp.

varyPorts If the source port in the outer UDP header should be varied, or be the same for each packet. Due to RSS packet distribution across RX-queues, varying this could improve performance. During the tests in this thesis, this parameter is always set to false, since there will always be just a single RX-queue used for T4P4S.

markTarget If this value is set to *True*, packets will be marked with an ID matching the target which will be responsible for processing this packet. A more detailed explanation can be found in section 4.1.3.

markerIDs If *markTarget* is set to *True*, this will contain an ordered list of which marker IDs should be inserted in which packets. This list is populated based on the specified workload share between T4P4S and sNIC, so all packets marked for a specific target also is included in the table configuration generated for that target.

5.1.2 Generating Table Entries

As explained in section 2.4, the control plane has to populate tables with correct rules. During this project, there were three different targets where tables had to be populated with rules matching each generated test scenario. These targets are *Barefoot Tofino*, *Netronome sNIC*, and *T4P4S*. Files generated for these targets contain the mappings between *inner_ipv4* destination addresses and *TEIDs*, so that the target P4 pipelines encapsulate packets correctly. The size of these lookup tables is a central part of this thesis, since the number of mapped TEIDs is a direct representation of the number of simultaneous users that each target is responsible for.

To populate tables in *Tofino*, a controller called *Open Network Operating System (ONOS)* has been used. Due to this target no longer being a part of the final thesis, there will be no need to spend time going in-depth in regards to that topic. ONOS is reading table entries in a JSON format, containing table rules as seen in the snippet presented in appendix A.3.1. There is a lot of overhead required for each rule; a lot of this information is for ONOS to recognize the device and application that should be updated.

The *Netronome sNIC* target is also reading table rules from a JSON formatted file, similar to ONOS. These files, much like those of ONOS, contain rules and default action for six tables, namely *dmac*, *firewall_DL*, *firewall_UL*, *smac*, *vEPG_UL*, and *vEPG_DL*. The first generated file is called *config_netronome_all.p4cfg*, and contains rules for all generated packets. Snippets from these configurations can be seen in appendix A.3.2. On top of this configuration file, separate files are generated containing rules for a subset of total generated packets to be used during offloading tests.

Configuration files are also generated for the *T4P4S* target. The controller which reads these files has been written specifically for this project, making the structure of these configuration files neat and simple. These are purely text files; each line containing space-separated key-value-pairs in a pre-defined order. For the encapsulation rules, the first column on every line is the *inner_ipv4* destination address in a *xxx.xxx.xxx.xxx* format, and the second column is the matching TEID. Snippets from the T4P4S configuration files are presented in appendix A.3.3.

For T4P4S, same as for the sNIC, configuration files are generated both for processing all generated packets, and separate files containing rules complementing the configuration files generated for the offloading sNIC; e.g. for 10% offloading, table configuration files are generated for the sNIC containing rules for 10% of packets, and configuration files for T4P4S are generated containing the other 90% of packets. Packets included in the 10% sNIC file are the packets which will be processed on the sNIC, i.e. the ones who are accelerated.

5.2 SmartNIC Setup

A Netronome Agilio CX 2x40G sNIC [22] was attached to an Ubuntu 18.04 Linux server, planned to act as a vEPG accelerator. The NFP Board Support Package (BSP)[2] and Software Development Kit (SDK)[3] were both installed on the host.

Since the Netronome Agilio CX series of network cards are complete self-contained systems, no optimizations on the host system has any impact on the performance of the sNIC. It does, however, require SR-IOV support for the NFP P4 pipeline to be able to send traffic directly down to the host user space applications.

The two 40G fiber ports on the sNIC were connected to a breakout module, each fiber splitting into 4x10G ports. Two of these new 10G ports were then connected to the NetFPGA running OSNT, which will act as traffic generator during these experiments.

5.3 DPDK Setup

DPDK 20.02 has been installed on the sNIC host server, and compilation of DPDK also compiles kernel modules for binding to the sNIC VF-ports.

An SR-IOV compatible *WS-C621E-SAGE* motherboard [3] equipped with a 10 core *Intel Xeon Silver 4114* CPU [13] running two threads per *2.2GHz* core and 2x16GB RAM clocked to *2133MHz* was used for DPDK processing and hosting the sNIC. *Linux kernel boot parameters* have been updated to optimize for the DPDK installation, as shown in listing 3.

These optimized kernel boot parameters make multiple changes, including isolating

[2]The NFP BSP released on 2018.06.29 was installed in the host system
[3]NFP SDK version 6.1.0.0 was installed in the host system

```
intel_pstate=disable mce=ignore_ce default_hugepagesz=1G hugepagesz=1G hugepages=6
    ↪ isolcpus=0,1,2,3 rcu_nocbs=0,1,2,3 nohz_full=0,1,2,3 iommu=pt intel_iommu=on
```

Listing 3: Kernel boot parameters optimized for this DPDK installation

the first four processor cores, preventing background processes from operating here; this way DPDK processing will be performed without any competition or context switching. These modified boot parameters also allocate more huge pages, enabling the Input–Output Memory Management Unit (IOMMU), and disable CPU scaling, all with the same goal of improved DPDK packet processing performance.

After a P4 program has flashed onto the sNIC, the pre-compiled *igb_uio* module is inserted in the Linux kernel. The DPDK module is then bound to the Netronome VF ports, removing the kernel network drivers which automatically bind to these devices. This way, the sNIC P4 pipeline can forward packets for continued processing in DPDK running on the host server.

5.3.1 Multiple Cores

The Netronome sNIC device in this setup is only reporting support of a single RX-queue while running P4 firmware; as explained in earlier in section 2.4.4, this results in DPDK only being able to utilize a single CPU core for packet processing. By installing a simple non-programmable 2x10G Intel x710 NIC and binding DPDK to this device, it has been verified that the multi-core limitation is due to issues specific to this sNIC device.

A workaround has been found to circumvent this problem by instead launching multiple identical DPDK applications bound to their own Netronome VF ports. The sNIC pipeline would then be responsible for load balancing traffic evenly across these VF ports, to share the workload among all DPDK processes.

The Netronome PMD guide does mention how the driver limitation will be solved in a future update[10]. Hence, no more time will be spent developing a workaround for this problem.

```
vEPG arch=dpdk hugepages=2048 model=v1model smem intelCores intelPorts
vEPG0 arch=dpdk hugepages=2048 model=v1model smem vf0_0 netronomeCores0 netronomePorts0
```

Listing 4: Snippet from T4P4S profile configuration file

```
intelCores  -> ealopts += -c 0x3 -n 4
netronomeCores0  -> ealopts += -c 0x1 -n 4
intelPorts  -> cmdopts += -p 0x3 --config "\"(0,0,0),(0,1,1),(1,0,0),(1,1,1)\""
netronomePorts0  -> cmdopts += -p 0x1 --config "\"(0,0,0))\""
vf0_0 -> ealopts += --proc-type=primary --file-prefix "a" -w 0000:b3:08.0
vf0_1 -> ealopts += --proc-type=primary --file-prefix "b" -w 0000:b3:08.1
```

Listing 5: Snippet from the T4P4S configuration, showing some aliases created for this project

Since the multi-core workaround solution would require a more complex DPDK controller and sNIC pipeline, a single DPDK core is used for the rest of this thesis.

5.3.2 T4P4S

When DPDK modules are bound to the sNIC VF ports, the T4P4S DPDK application must also compile and launch behind these ports. Multiple T4P4S profiles have been created, each one for some specific test requirements; listing 4 show a snippet from the T4P4S profile configuration file, including two of the profiles used during this project.

Aliases used in these profiles have been specifically created for this thesis project. Some of the T4P4S aliases created for this project can be found in listing 5. The first two aliases specify which cores should be used to run the T4P4S DPDK application, as well as how many memory channels should be used. The last two aliases specify the mapping between ports, queues, and cores.

'*-p 0x3*' is a bit string specifying which ports the application should listen to;

'*–config*' takes a list of tuples specifying port, queue, core mappings in a *(port,queue,core)* format;

'*netronome0*' tests listen to the first port and then process incoming packets on a single CPU core;

'*intel*' tests listen to the first two ports, each port distributing packets to the same two RX-queues, and one port being responsible for processing packets in each RX-queue;

'*vf0_0*' and '*vf0_1*' are used while creating multiple simultaneous T4P4S instances, each bound to a different VF port. These aliases were created for the multi-core workaround explained in section 5.3.1.

On top of this, a few changes had to be made to the T4P4S code base for this application. By default, the P4 match-action tables are implemented as hash-tables in C; the maximum size of these hash-tables is *1024* entries, which was increased to *102400* for the back end hash tables to be able to store all TEID mappings used in this thesis.

Before transmitting packets, DPDK is storing these packets in a buffer which will then be sent as a batch; this batch is sent either when 32 packet are received, or after a certain timeout duration. By default, T4P4S has set this timeout duration to $100us$, which will greatly increase packet latencies while the system is under low load; it has therefore been reduced to $10us$.

5.3.3 T4P4S Controller

Before starting the T4P4S process, it is first necessary to load the T4P4S controller process, which has been written in C specifically for this project, and is used to populate three P4 tables: *portfwd*, *firewall_DL*, and *vEPG_DL*.

portfwd and *firewall_DL* are both populated with the same 1000 static entries each time, as read from files. *portfwd* reads a file with each line containing two integer values, which represent *ingress_port* and *egress_port*, and are written as-is to the portfwd table in the P4 program. This table only contains a single entry, which will forward all packets from port 0 to 1. Listing 6 is the complete file used to specify this rule, which can easily be expanded to allow more complex packet forwarding functionality in the T4P4S pipeline.

The vEPG also includes a firewall, where it can be instructed to drop packets based on specified IP addresses; for this pipeline implementation, the list of IP addresses is read

```
0 1
```

Listing 6: T4P4S port forwarding table configuration file, forwarding all packets received through port 0 to port 1

```
87.179.138.75
233.60.20.29
111.152.239.22
207.157.186.112
```

Listing 7: Snippet from T4P4S IP firewall table configuration file

from a text-file, where each line contains an IP address as explained earlier in section 5.1.2. The firewall configuration file has been generated so that none of the addresses listed will ever be present in the packet traces used during the tests, to ensure consistency across tests. Listing 7 show a few lines from the firewall file, demonstrating the simple format used for IP firewall configuration in the T4P4S controller for this project.

While the T4P4S controller is reading the firewall configuration file, it splits each line into four 8bit values around the 'dots', and stores these in a global array containing all parsed IP addresses; after the full file has been parsed, all rules are inserted into the T4P4S P4 firewall table one at a time. All tables rule configurations are processed in a similar way, where they are read as a batch from a configuration file, and then inserted as rules one at a time to the correct T4P4S P4 table.

In contrast to the two previous configuration files, the file containing *vEPG_DL* rules is generated specific for each test case, populating the encapsulation table with rules specifying GTP encapsulation for the packets which will be encountered during specific tests. Varying the size of the GTP encapsulation table is an important part of the project, since it is a direct representation of how many GTP tunnels are allocated for processing in this target. The *vEPG_DL* file contains a list of IP addresses together with an integer value ranging from 0 to $2^{31}-1$, specifying which inner_IP destination address should be encapsulated with which TEID value. Entries in this configuration file are generated alongside the

```
205.212.147.22 644680905
37.181.194.235 604547847
88.139.83.242 765561505
237.164.169.14 1578545253
```

Listing 8: Snippet from T4P4S GTP encapsulation table configuration file, showing generated IP/TEID mappings

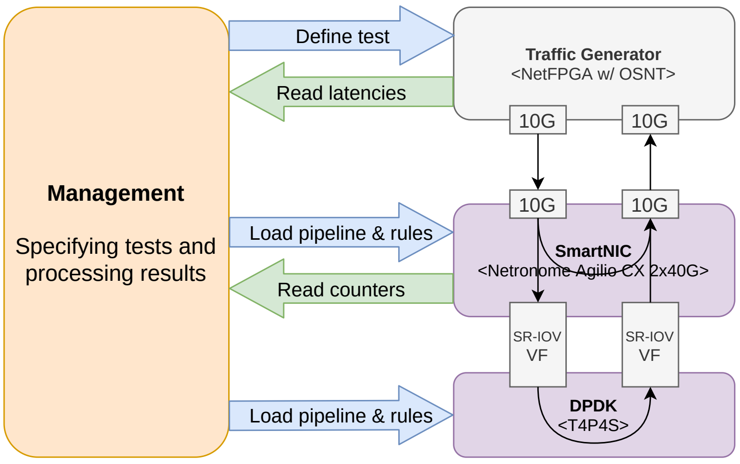


Figure 5.1: Testbed design

packet traces, to ensure all packets which T4P4S will receive during the test has a matching rule inserted in the GTP encapsulation table. IP addresses in this file are processed the same way as before for the firewall table, but also include the mapped TEID alongside it. Listing 8 show the first few lines from one of these files; note how the rules are generated according to the *flows.json* file mentioned in section 5.1.

After the controller has been started, a T4P4S data plane process is launched; the process will attach to the controller, which in turn will populate the pipeline match-action tables as explained above.

5.4 Testbed

Setting up a testbed for measuring the impact of sNIC offloading was a substantial part of the project. The result is a system for fully automated experiments, requiring only an

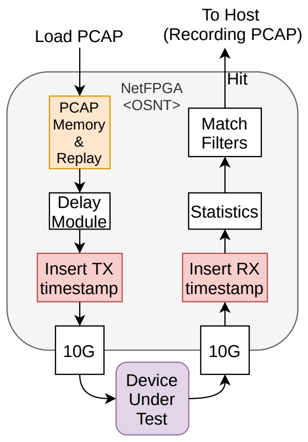


Figure 5.2: OSNT simplified explanation. Single TX to another RX port

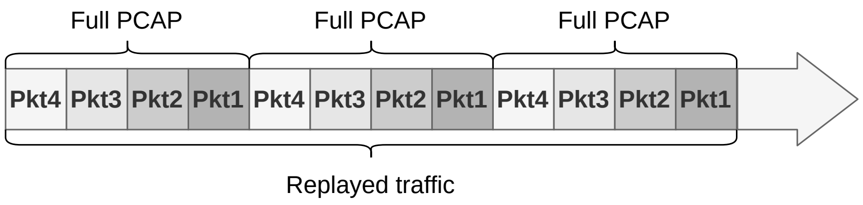


Figure 5.3: OSNT replaying a pre-generated packet trace, containing four packets, three times

initial list of testing parameters specifying tests to perform and record as a batch. An overview of experimental procedure can be found in section 5.4.2.

5.4.1 Traffic Generation - OSNT

A *NetFPGA SUME*[8] running *OSNT*[2] is used as traffic generator during these experiments. OSNT is a powerful FPGA hardware traffic generator with support for packet latency measurements. A very simplified visualization how for OSNT works is presented in figure 5.2.

As previously described in section 5.1.1, the traffic generator emits network traffic according to pre-generated PCAP-files containing packet traces; packets in these PCAP-files emits in-order as seen in figure 5.3. This way, all flows will be of equal intensity, and could provide difficulties for cache-based optimizations in each target, since flow packets

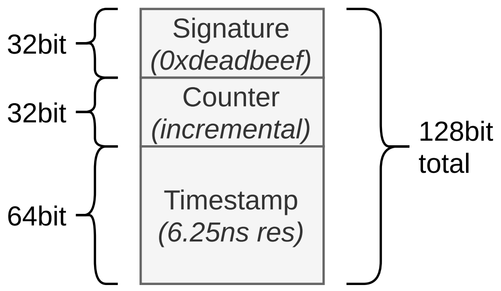


Figure 5.4: Structure of OSNT timestamp marker as written to packets

never appear back-to-back; shuffled packet transmission orders, and varied flow intensities, should be included in future tests.

Timestamps can be inserted both during packet transmit and receive; the timestamp difference will show the duration each packet has spent in the network before its return to the NetFPGA. An offset for where in the packet to insert timestamps has to be specified, so as not to overwrite any essential headers. Two approaches have been used during this project; prepending the timestamp before the first ethernet header, and placing it in packet payload. These timestamps have an impressive 6.25ns resolution[4].

The OSNT timestamp module also inserts an OSNT signature of *0xdeadbeef*, which can be used to detect the timestamp, as well as an incrementing packet counter, as seen in figure 5.4. Prepending an OSNT header to packets should not impact the evenness of the RSS queue distribution, described earlier in section 2.4.4; as long as the RSS hash includes at least the packet counter as input, enough entropy should be included in RSS hash calculation to achieve an even queue distribution.

OSNT can be instructed to emit packets at a specified *Inter-Packet Gap (IPG)*, which is a delay in between the start of each packet transmission. If the PCAP-file only contains packets of the same size, the IPG would be a viable tool for specifying the bit rate of generated traffic. There is also an additional rate limiter module built into OSNT; due to the restricted testing of a single packet-size at a time, specifying IPG was enough in regards

[4]For reference, light itself can only travel about 1.9 meters during 6.4 nanoseconds. This means that we might even calculate the speed of light itself using this setup, by varying the length of fiber cables

to this project. Due to encapsulation inserting extra headers to packets, thereby increasing their respective sizes, the RX- and TX bit-rates will not be identical. For downlink traffic (coming from DCGW to eNodeB), encapsulation results in each packet increasing in size by 36 bytes, resulting a higher bit-rate exiting the device than is received, while the reverse is occurring for uplink traffic, where packets decrease in size by 36 bytes. During these tests, the IPG is calculated to ensure that the specified bit-rate is reached at the most loaded link; it is therefore calculated as shown below in equation 5.1.

$$IPG = \begin{cases} \frac{8*(pktSize+36)}{bitrate}, & \text{if downlink} \\ \frac{8*pktSize}{bitrate}, & \text{if uplink} \end{cases} \tag{5.1}$$

Furthermore, there is another noteworthy *statistics* module built into OSNT which is calculating simple statistics based on incoming network traffic. During the course of these experiments, the values extracted from this statistics module are the *Packets Per Second (PPS)* and packet-counter values. The packet counter is incrementing each time a packet is received by the NetFPGA. It is therefore possible to calculate the total system packet loss as shown below in equation 5.2.

$$loss_{total} = replays * PCAP_{numPkts} - OSNT_{RX} \tag{5.2}$$

PPS measurements are also extracted using this aforementioned statistics module. Scripts on the NetFPGA host are polling the NetFPGA for these statistical values once per second, until there is no more traffic detected. The last measurement is removed from the stored results, and a mean value together with standard deviation is recorded based on values polled from the NetFPGA. The reason for omitting the last measured PPS from results is due to traffic not flowing for the full second, resulting in the last value being invalid.

An integer specifying how many times to replay the trace has to be set while initiating traffic generation. For most of these tests, this was calculated so that traffic would be generated for a total of 10 seconds to ensure that enough PPS measurements are retrieved. The number of times the PCAP should be replayed in OSNT was calculated as shown below in equation 5.3, where $time = 10s$.

$$numReplays(time) = \left\lceil \frac{bitrate * time}{8 * pktSize * PCAP_{numPkts}} \right\rceil \tag{5.3}$$

It is preferred to split latency measurements based on processing target target, to enable measuring what impact the offloading design has on packets processed by each target. Splitting measurements based on processor is achieved by using the *filter* module of OSNT. This module determines which packets should be sent from NetFPGA down to CPU for further analysis. The actual timestamp differences are calculated in CPU, which means that this filter will determine which packets will be included in the latency measurements.

During encapsulation, each target modifies the outer IP header of packets so that it will point to the DCGW. By having each target instead encode its own unique destination IP address in this header, the value can be specified in the OSNT filter so that only packets processed by a specific system are included in latency measurements on the NetFPGA host. Since this measurement splitting method requires the NetFPGA to parse the packet IP header, it is incompatible with a prepended OSNT header. Due to this, OSNT instead inserts timestamping information into packet payloads.

Packet encapsulation is performed in the Device Under Test (DUT), which changes the offset within each packet where the payload is starting. Because of this, the analysis scripts running on the traffic generator parse the timestamping data with an offset of 36 bytes compared to where they are inserted, to compensate for headers inserted during encapsulation. During measurements where the DUT is instead decapsulating packets, the offset is set to -36 bytes.

Each experiment is performed up to three times, with three different filters. The first test allows all packets down to CPU, the second only allows sNIC processed packets, and the last time only allowed DPDK processed packets. This way it is possible to analyze the performance of individual sub-processor during each test scenario.

5.4.2 Management Server

A server was set up to remotely control all machines involved in the experiments. This manager is fed a list of testing parameters, and iterates over these until completion. For each combination of inserted parameter values, the manager is loading required components and instructions onto all involved machines. For each test, this is the procedure executed by the manager:

1. Reset devices from last test
2. Start sNIC and flash new firmware
3. Populate match-action tables in sNIC
4. Bind DPDK drivers to sNIC VF ports
5. Start T4P4S controller, prepared with table entries
6. Compile and start T4P4S P4 pipeline
7. Load packet trace onto NetFPGA
8. Perform single trace replay at high IPG to populate device caches
9. Reset counters in sNIC and OSNT
10. Replay packet trace from NetFPGA
11. Parse PCAP containing subset of test packets, extracting latencies
12. Download latency histogram from traffic generator

13. Store test parameters and results in Comma-Separated Values (CSV)

5.5 Maximum Capacity Definition

A definition of what is meant by the maximum packet processing rate of a target has to be established. Following the RFC2544 standard[5], frame loss rate is defined as $\frac{100(input\_count-output\_count)}{input\_count}$. $output\_count$ is here defined as $replays * PCAP_{numPkts}$, and $input\_count$ is an internal RX-counter included in the statistics module of OSNT.

Maximum device packet processing rates can be defined in multiple ways; below are two relevant definitions based on the packet loss rate.

Non Drop Rate (NDR) does not allow for any packet loss while the device is under maximum packet processing load. This definition is problematic, since even a single packet loss caused by faulty drivers could greatly impact the maximum packet processing rate. As seen later in section 6.1.1, T4P4S has a problem where a very infrequent packet loss starts already at a moderate load; basing the maximum packet processing rate on this definition might therefore be misleading.

Partial Drop Rate (PDR) allows for a partial drop rate to occur while the device is under maximum load. The T4P4S partial drop rate during moderate load is very low, often under 0.01%. The industry standard is to allow a 0.5% loss rate for PDR throughput, and is what is used in this thesis.

The maximum packet processing rate of a device is now defined as R^+, which is the maximum possible packet arrival rate where the packet loss rate is less than 0.5%.

6 Evaluation

Performance of the hybrid VNF processor, performing basic layer 3 firewall functionality and GTP processing, together with the performance of both individual sub-processors is

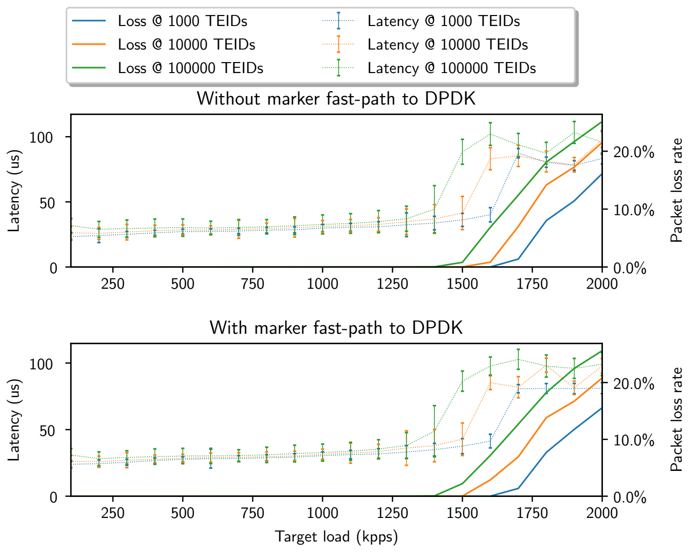


Figure 6.1: T4P4S performance while performing GTP encapsulation of 128B packets

measured and presented in this section.

During all these tests, GTP encapsulation is performed as explained in section 4.2.5; the same sub-processor performing GTP encapsulation will also apply a layer 3 firewall table populated with 1000 IP-matching rules, none of the addresses in this table will be present in the replayed packet trace. GTP processing and basic layer 3 firewall functionality are therefore both accelerated at the same time.

A single 10G fiber port is used for sending traffic from OSNT to the DUT, with a second port used for sending traffic back from DUT to OSNT after processing; the testbed is explained in greater detail earlier in section 5.4. An OSNT timestamp is inserted in the payload of every packet, and is used for latency measurements during all tests, as explained in section 5.4.1.

6.1 T4P4S vEPG Performance Behind SmartNIC

To evaluate the performance benefit gained by offloading certain flows to sNIC, base T4P4S vEPG measurements without any active sNIC offloading has to be performed. During these measurements, DPDK is still bound to the sNIC VF ports as described in section 4.1.2. Two different offloading pipelines exist for the sNIC, as explained in section 4.1; two equivalent tests have been performed with T4P4S running behind both of these sNIC pipelines, one at a time, without populating any offloading sNIC tables. Packet traces containing 128B downlink packets are replayed through the hybrid non-offloading target, with load increasing at $100KPPS$ steps to measure performance at various target loads.

Measurements recorded during these tests are presented in figure 6.1. According to these measurements, our T4P4S instance appears to handle loads of up to $1.6MPPS$ relatively consistently; packet latencies appearing to be dependent on target load as expected. When the target load is increased above certain thresholds, packet loss skyrockets and the overall packet latency increases significantly. Number of mapped TEIDs appear to impact the target performance, which is unexpected seeing as these mappings are stored in hash tables; explanations to this phenomenon could be either reduced cache efficiencies, or increased length of linked lists in backend hash table where these mappings are stored. Based on the measurements presented in figure 6.1, there is no significant difference between the two offloading designs in terms of performance; the marked approach should, according to these results, therefore be a viable option when more complex NFV architectures are preferred as explained earlier in section 4.1.3.

6.1.1 Early Packet Loss

While verifying functionalities of various components used in this project, an unexpected early drop rate was detected in flows processed by the T4P4S target while running behind the offloading sNIC pipeline. To find out if the sNIC system caused this phenomenon, two equivalent low-load tests are performed similarly to earlier in section 6.1, with DPDK bound to either a non-programmable Intel x710 2x10G NIC, or the sNIC executing the non-marker pipeline without any offloading enabled.

Early drop rate was still present even while binding DPDK to the non-programmable

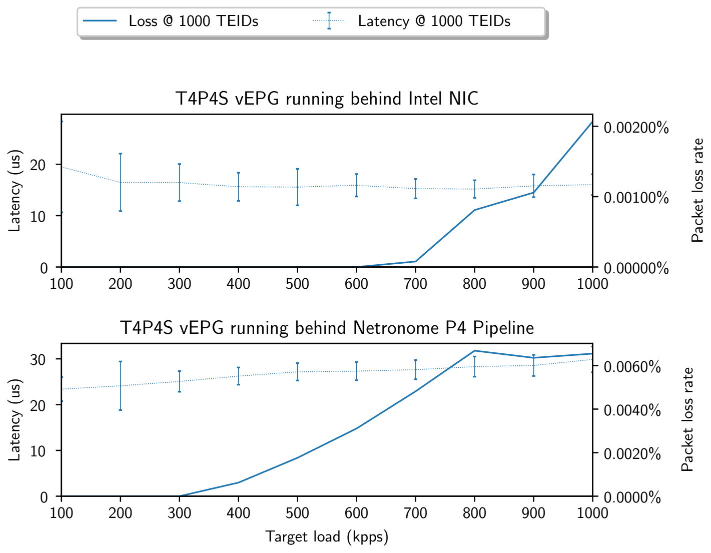


Figure 6.2: T4P4S loss rates while performing GTP encapsulation, 0% offloading, marker enabled. Early packet loss before maximum capacity

NIC, as shown in figure 6.2. Packet loss, measured as explained in section 5.4.1, was very infrequent but still a consistent issue both with DPDK bound to the sNIC and while bound to the non-programmable Intel 2x10G NIC, although even more infrequent behind the non-programmable NIC.

The T4P4S application, as it was set up and configured for this project, had an issue with early packet loss. Figure 6.2 presents calculated packet loss rates during low-load scenarios, showing infrequent but consistent packet loss at processing rates much lower than the maximum T4P4S capacity as measured later in section 6.1.2. To confirm that this was not an issue caused by the offloading sNIC setup, equivalent tests was performed while binding DPDK to Intel x710 NIC ports; the same early loss issue was present there as well, although not as early and frequent. Due to this early loss issue, maximum packet rate has to be defined based on PDR, as explained in section5.5, where packet loss below a certain threshold is allowed while the DUT is under maximum load.

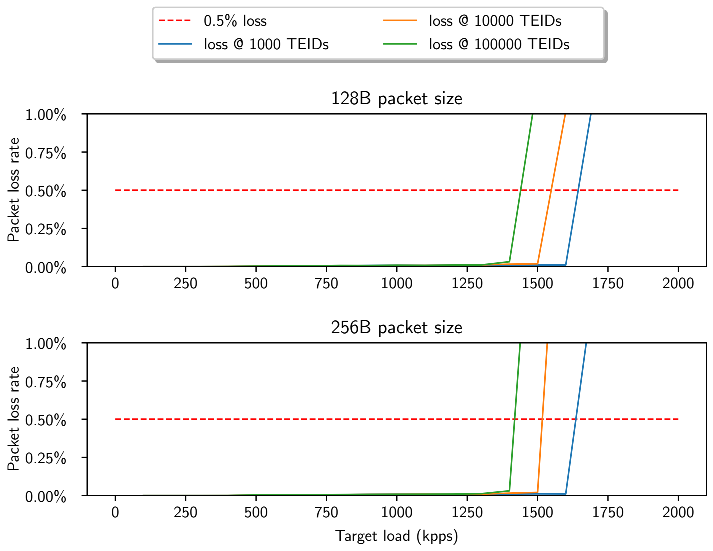


Figure 6.3: T4P4S loss rates while performing GTP encapsulation behind SmartNIC, 0% offloading, marker enabled

Table 6.1: Maximum T4P4S packet rates while performing GTP encapsulations behind SmartNIC, 0% offloading, marker enabled

Num TEIDs	Max Rate
1000	$1.6MPPS$
10000	$1.5MPPS$
100000	$1.4MPPS$

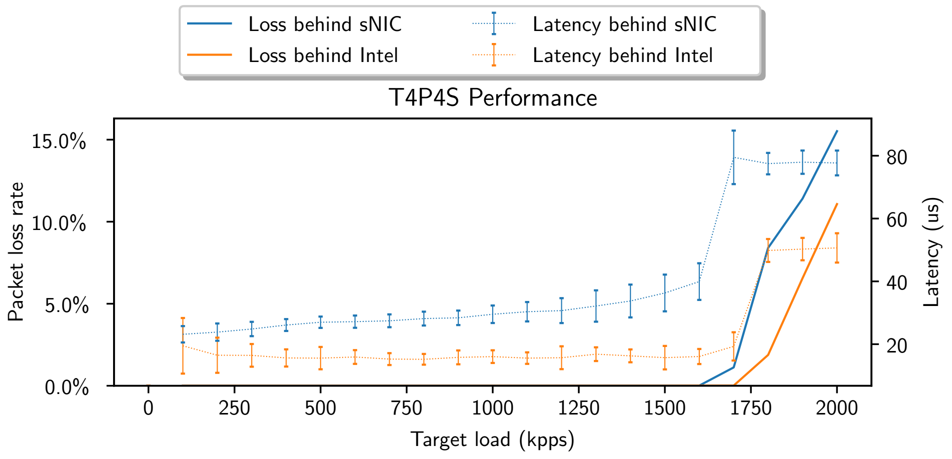


Figure 6.4: T4P4S performance while performing GTP encapsulation of 128B packets, comparing sNIC impact on performance. sNIC offloading is disabled during sNIC measurements. 1k mapped TEIDs

6.1.2 Maximum Packet Rate

The maximum packet processing rate for the T4P4S pipeline implementation has to be measured. DPDK is bound to sNIC VF ports; the sNIC is running the marker pipeline, as explained in section 4.1.3, with offloading disabled. The number of mapped TEIDs impact the T4P4S performance; because of this, multiple tests are performed while varying the number of mapped TEIDs (1k, 10k, 100k). Packets are sent at an increased rate in $100kpps$ steps, until the measured packet loss exceeds the 0.5% threshold, as explained in section 5.5.

Measured loss rates are presented in figure 6.3 for *128B* and *256B* packets, at increasing packet rates. It is shown that the number of mapped TEIDs impact the maximum processing rate of the T4P4S target; maximum packet processing rates, prior to reaching the 0.5% loss rate threshold, are presented in table 6.1. Packet size did however not have an impact on the maximum processing rate during these tests, which confirms that the bottleneck is due to processing overheads.

6.2 T4P4S - Latency Imposed by SmartNIC Overhead

An offloading sNIC running a P4 pipeline imposes an additional delay on traffic destined for T4P4S processing, which has to be measured. To measure this imposed latency, the T4P4S process was bound to a traditional 2x10G Intel NIC, model x710, instead of the offloading sNIC, where performance measurements were performed. During these tests, the non-marker design was running on the sNIC with offloading disabled.

Packet processing performance for T4P4S mapping 1k TEIDs behind either the non-programmable 2x10G Intel NIC or the offloading sNIC pipeline is presented in figure 6.4, showing an approximately $14us$ additional base latency - on top of the $15us$ latency measured behind Intel - imposed by the sNIC pipeline without active offloading. During earlier measurements, a base latency of $6us$ for the Netronome Agilio CX 2x40G was found; seeing how packets have to pass through the Netronome P4 pipeline twice in this setup, a minimum imposed latency of $12us$ was expected.

It is likely possible to significantly reduce this additional latency imposed by the offloading sNIC pipeline. One such approach could be to modify the Micro-C code generated by the Netronome P4 compiler, and letting packets ingressing from T4P4S completely bypass the P4 pipeline, or letting T4P4S egress through a non-programmable NIC. However, since this thesis is just a proof of concept, implementing a basic prototype, the system set up for these experiments is enough to prove a performance boost gained by sNIC offloading.

The additional base latency imposed by the offloading sNIC should be the same for all offloading cases, not dependent on the complexity of the accelerated VNF; it is therefore reasonable to expect greater performance benefits gained by sNIC acceleration in more complex NFV environments where more processing overhead is present.

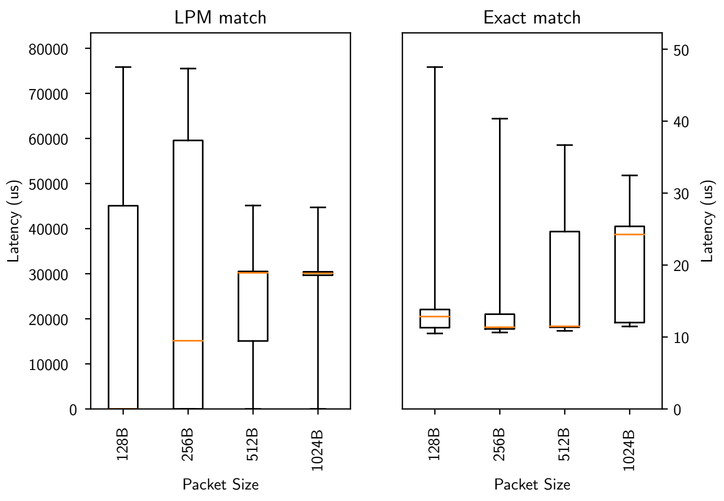


Figure 6.5: Netronome latencies while mapping 100k TEIDs at 1Gbps. Impact of LPM matching

6.3 SmartNIC Performance

For the Netronome Agilio CX 2X40G sNIC to be evaluated as a viable VNF accelerator, it is necessary to measure its stand-alone packet processing performance to ensure it can deliver satisfactory results; the sNIC VNF implementation has therefore been performance tested in isolation, with this section presenting findings from these tests.

6.3.1 Performance of LPM Table Matching

In the code which the sNIC P4 pipeline was based upon performs Longest Prefix Match (LPM) matches against IP addresses to find the mapped TEID. Since every TEID has at most a single IP encapsulated by it in the scenario evaluated in this thesis, LPM lookup is not required for performing this GTP functionality.

As seen on the left hand side in figure 6.5, storing a TEID behind an LPM match result in extremely high packet latencies. On the right hand side in figure 6.5, the performance of storing these mappings behind exact-matches is presented. This simple pipeline

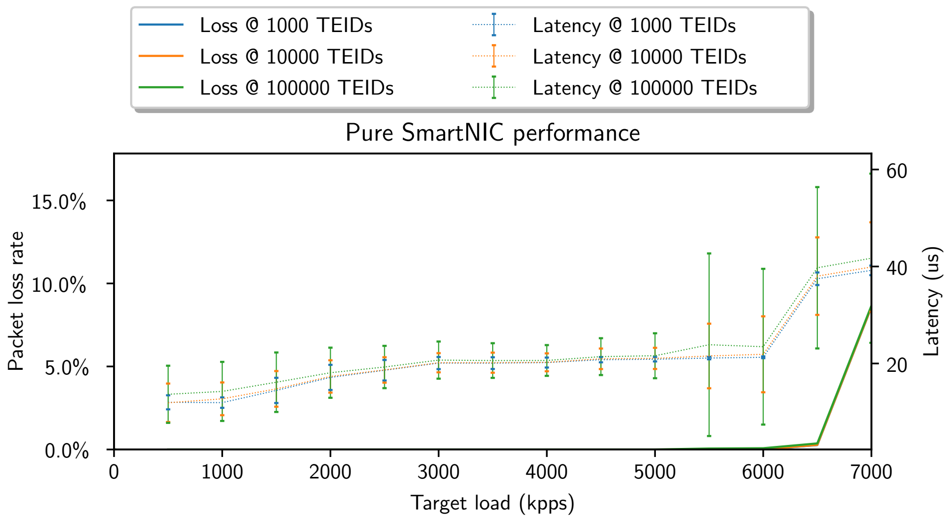


Figure 6.6: SmartNIC stand-alone performance while processing 128B packets

modification greatly improved the sNIC GTP performance, reducing the packet processing latencies to less than a thousand of what they were before. All designs in this project will as a direct consequence of this be using exact matching for all TEID mappings.

6.3.2 SmartNIC Stand Alone vEPG Performance

The performance of the offloading SmartNIC as a stand-alone vEPG processor has to be measured for its behavior to be understood, which is here accomplished by instructing the device to offload 100% of traffic; thereby including eventual offloading overhead in the results.

Packet traces containing 128B downlink packets was replayed through the sNIC, configured for 100% offloading, while increasing traffic load at $500KPPS$ steps; results from these tests is presented in figure 6.6.

According to these measurements, the sNIC appears to handle vEPG downlink processing of 128B packets up to $5MPPS$ without performance degradation, and up to $6MPPS$ without any significant packet loss. A low latency jitter is delivered, with average latencies ranging from $10us$ during low load to $20us$ at a higher load. The Netronome sNIC delivers

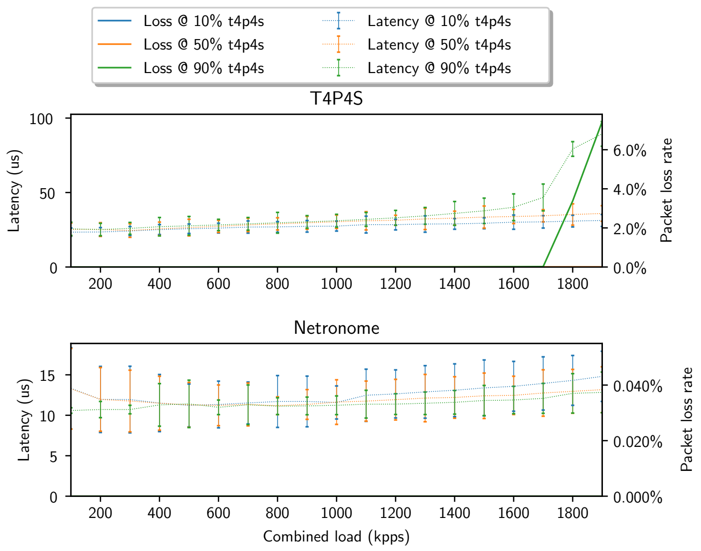


Figure 6.7: Netronome and T4P4S performance while sharing 10K TEID workload, with marker approach for SmartNIC offloading

vEPG functionality at lower packet latencies than the T4P4S system managed even at the best scenarios while bound to an ordinary Intel NIC. Combined with significantly reduced latency jitter, the Netronome sNIC appears to be a viable VNF accelerator in this case.

6.4 Hybrid Target vEPG Performance

6.4.1 Offload Percentage Impact on Accelerated Traffic

For maximizing total network performance, one might want to accelerate as many traffic flows as the accelerator can handle. However, does maximum sNIC offloading have a negative impact on the acceleration benefit of each individual accelerated flow? Performance of both sub-processors where varying percentages of total traffic is offloaded has therefore been measured.

Figure 6.7 presents measured performance of both sub-processors in the hybrid vEPG target, where varying percentages of total traffic is accelerated by the sNIC. The optimized encapsulation pipeline was deployed, packet traces with 128B downlink packets are

Table 6.2: Test parameters used to measure hybrid target performance while keeping T4P4S at constant 1500 KPPS load

Tapas Share	Total Load (KPPS)	sNIC Load (KPPS)	T4P4S Load (KPPS)
100	1500	0	1500
90	1666	166	1500
80	1875	375	1500
70	2142	642	1500
60	2500	1000	1500
50	3000	1500	1500
40	3750	2250	1500
30	5000	3500	1500

replayed, while a single CPU core was used for T4P4S processing.

According to the results in figure 6.7, the performance benefit of sNIC acceleration is dependent on how much of total traffic is offloaded to the sNIC; i.e. when a larger percentage of total traffic is offloaded, the performance benefit gained from acceleration decreases. It might therefore be beneficial to only accelerate traffic which actually requires it, leaving less time-sensitive traffic flows to be processed on the host as usual; this way maximizing the performance benefit gained by accelerating these flows.

6.4.2 Combined Aggregate Performance

In this section, performance of the new hybrid target will be evaluated. To achieve this, the T4P4S sub-processor will be kept at its maximum capacity, as presented earlier in section 6.1.2; while T4P4S is under maximum load, the packet processing load on the offloading sNIC is gradually increased.

Due to the hybrid system design requiring a shared incoming packet stream, the test parameters are set according to table 6.2 to achieve a gradually increased load on the sNIC with constant T4P4S load at maximum capacity.

To perform these tests using the packet marker approach, new traces have to be generated with correct individual marker values. Traces have been pre-generated with offloading

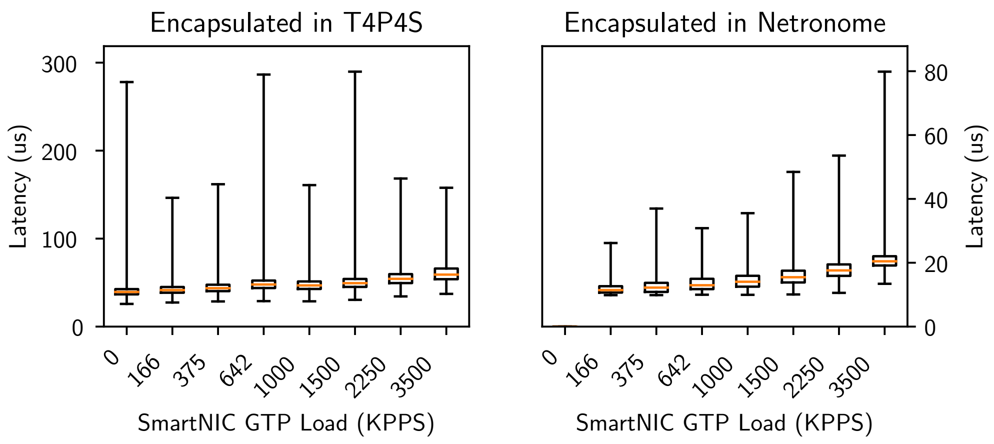


Figure 6.8: Latency comparison for packets encapsulated in T4P4S or Netronome. 128B packets, in total 10k mapped TEIDs, marker offloading design

percentages in 10% increments, starting at 0% up to 100% in each target. Because of this, the total load required to keep T4P4S at constant load for each of these offload percentages was calculated according to the following formula: $TotalLoad = \frac{TapasTarget}{TapasPercent}$

Figure 6.8 presents recorded latencies during tests using the load parameters from table 6.2, where the combined hybrid target process a total of 10k TEIDs.

Even while the sNIC is forwarding traffic to T4P4S at the T4P4S maximum capacity, the sNIC still manages to deliver lower latencies and higher consistency compared to the T4P4S sub-processor. The Netronome sNIC also managed these results without dropping a single packet assigned to the sNIC, making the sNIC a reliable packet processor.

A clear improvement both in terms of latency and jitter for offloaded flows can be seen in figure 6.9, where a combined load of *3000 KPPS* is sent to the hybrid processor, and 50% of traffic is offloaded to the sNIC for processing.

The testbed design used during these experiments is not able to generate traffic at a high enough intensity to completely saturate the sNIC, which means that a more advanced setup is required for measuring the maximum aggregated capacity of this new hybrid target; that would require a traffic generator to utilize multiple fibers in parallel for delivering network

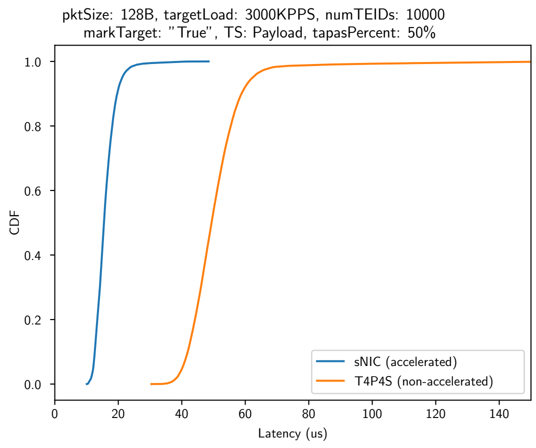


Figure 6.9: Netronome and T4P4S processing latencies while sharing 10K TEID combined workload, with marker approach for SmartNIC offloading. Latency CDF during 3000 KPPS 128B load with 50/50 offloading hybrid system

traffic to the hybrid vEPG target.

The testbed used during these measurements did manage to confirm successful vEPG processing at a rate of at least $6500KPPS$ by the sNIC alone, which is already a significant improvement in total capacity compared to T4P4S as a stand alone vEPG processor.

7 Conclusion

A hybrid system performing GTP encapsulation and firewall functionality in a mobile packet core network has been designed and implemented, with a T4P4S compiled DPDK application and equivalent Netronome Agilio CX 2x40G sNIC pipeline sharing total workload. A certain percentage of packets assigned to this hybrid target can be offloaded to the sNIC, thereby providing acceleration for time-sensitive network traffic flows.

In this proof-of-concept implementation, traffic flows accelerated by the offloading sNIC has half as much latency compared to traffic processed on the equivalent DPDK T4P4S pipeline running on host CPU, while at the same time significantly reducing latency jitter. Measured latencies for packets processed in T4P4S were in the range of $20us - 45us$

depending on test scenario, while packets marked for sNIC acceleration measured in the $10us - 24us$ range.

These results indicate that sNIC offloading can be used to increase network performance for time-sensitive URLLC flows, while at the same time allowing less time-sensitive flows to be processed in an NFV environment on commodity server hardware.

Static performance evaluations, as used in this thesis, is not a perfect representation of a real world mobile packet core scenario. To ensure hybrid packet processor viability in handling URLLC flows, its performance in a dynamic environment has to be evaluated; including varied flow intensities, burst patterns in network traffic, live table rule updates, etc. A more complex test bed than what was used here is required to fully saturate the sNIC, and thereby measuring the total aggregate capacity of this hybrid target. The hybrid design, as designed and implemented in this project, is capable of offloading more varied VNFs than what was evaluated in this thesis. A more general evaluation of offloading specific operations is desired, including pure computationally intense operations used in artificial intelligence, to parser-heavy Deep Packet Inspection (DPI) functionality useful in complex firewall systems; thereby opening up for the possibility of providing even these complex security-enhancing services for URLLC traffic flows.

References

[1] Sherif Abdelwahab et al. “Network function virtualization in 5G”. In: *IEEE Communications Magazine* 54.4 (2016), pp. 84–91.

[2] Gianni Antichi et al. “OSNT: Open source network tester”. In: *IEEE Network* 28.5 (2014), pp. 6–12.

[3] ASUS. *WS C621E SAGE — ASUS USA*. June 4, 2020. URL: https://www.asus.com/us/Commercial-Servers-Workstations/WS-C621E-SAGE/specifications/.

[4] Pat Bosshart et al. “P4: Programming protocol-independent packet processors”. In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014), pp. 87–95.

[5] Scott Bradner and Jim McQuaid. *RFC2544: Benchmarking Methodology for Network Interconnect Devices*. 1999.

[6] Margaret Chiosi et al. “Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action”. In: *SDN and OpenFlow world congress*. Vol. 48. sn. 2012, p. 202.

[7] Andrea Detti. “Functional architecture”. In: *CNIT-Electronic Eng. Dept., Université de Rome Tor Vergata* ().

[8] Digilent. *NetFPGA-SUME [Reference.Digilentinc]*. June 2, 2020. URL: https://reference.digilentinc.com/reference/programmable-logic/netfpga-sume/start?redirect=1.

[9] Chetan Hiremath John Mangan Michael Lynch DongJin Lee JongHan Park. *Towards Achieving High Performance in 5G Mobile Packet Core’s User Plane Function*. Tech. rep. Intel, 2018.

[10] DPDK. *NFP poll mode driver library*. URL: https://doc.dpdk.org/guides/nics/nfp.html (visited on 05/18/2020).

[11] Xenofon Foukas et al. “Network slicing in 5G: Survey and challenges”. In: *IEEE Communications Magazine* 55.5 (2017), pp. 94–100.

[12] Zhanwei Hou et al. “Ultra-reliable and low-latency communications: prediction and communication co-design”. In: *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE. 2019, pp. 1–7.

[13] Intel. *Intel Xeon Silver 4114 Processor Product Specifications*. June 4, 2020. URL: https://ark.intel.com/content/www/us/en/ark/products/123550/intel-xeon-silver-4114-processor-13-75m-cache-2-20-ghz.html.

[14] DPDK Intel. *Data plane development kit*. 2014.

[15] Hyoungju Ji et al. “Introduction to ultra reliable and low latency communications in 5G”. In: *arXiv preprint arXiv:1704.05565* (2017).

[16] Georgios P Katsikas, Gerald Q Maguire Jr, and Dejan Kostić. “Profiling and accelerating commodity NFV service chains with SCC”. In: *Journal of Systems and Software* 127 (2017), pp. 12–27.

[17] Tamás Lévai et al. “The price for programmability in the software data plane: The vendor perspective”. In: *IEEE Journal on Selected Areas in Communications* 36.12 (2018), pp. 2621–2630.

[18] Yong Li and Min Chen. “Software-defined network function virtualization: A survey”. In: *IEEE Access* 3 (2015), pp. 2542–2553.

[19] Leonardo Linguaglossa et al. “Survey of performance acceleration techniques for network function virtualization”. In: *Proceedings of the IEEE* 107.4 (2019), pp. 746–764.

[20] Guyue Liu et al. “Design challenges for high performance, scalable nfv interconnects”. In: *Proceedings of the Workshop on Kernel-Bypass Networks*. 2017, pp. 49–54.

[21] Netcope. *P4 to VHDL*. 2019. URL: https://www.netcope.com/en/products/p4-to-vhdl (visited on 05/14/2019).

[22] Netronome. *Agilio CX SmartNICs - Netronome*. May 7, 2020. URL: https://www.netronome.com/products/agilio-cx/.

[23] Netronome. *NFP-4000 Theory of Operation*. URL: https://www.netronome.com/m/documents/WP_NFP4000_TOO.pdf (visited on 05/14/2019).

[24] Netronome. *Programming Netronome Agilio® SmartNICs*. URL: https://www.netronome.com/m/documents/WP_NFP_Programming_Model.pdf (visited on 05/14/2019).

[25] Barefoot Networks. *Product Brief Tofino*. May 7, 2020. URL: https://barefootnetworks.com/products/brief-tofino.

[26] Rolf Neugebauer et al. “Understanding PCIe performance for end host networking”. In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 2018, pp. 327–341.

[27] Paul Quinn, Uri Elzur, and Carlos Pignataro. “Network service header (NSH)”. In: *RFC 8300*. RFC Editor, 2018.

[28] Ashok Sunder Rajan et al. “Understanding the bottlenecks in virtualizing cellular core network functions”. In: *The 21st IEEE International Workshop on Local and Metropolitan Area Networks*. IEEE. 2015, pp. 1–6.

[29] Filipo Sharevski. “Towards 5G cellular network forensics”. In: *EURASIP Journal on Information Security* 2018.1 (2018), p. 8.

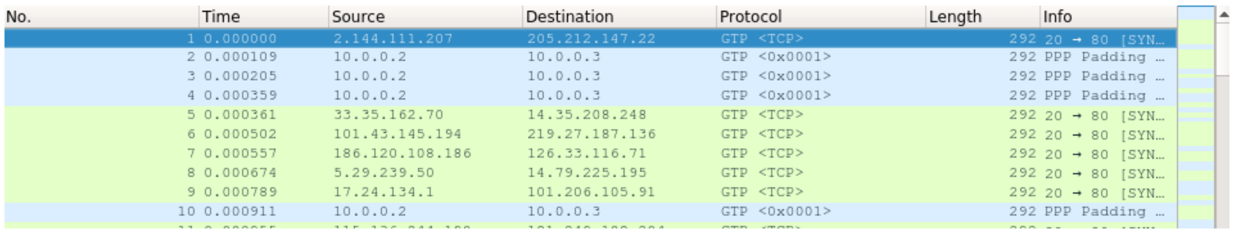
[30] Myung-Ki Shin et al. “A way forward for accommodating NFV in 3GPP 5G systems”. In: *2017 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE. 2017, pp. 114–116.

[31] 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.* 2019, pp. 9–14.

[32] John Thompson et al. “5G wireless communication systems: Prospects and challenges [Guest Editorial]”. In: *IEEE Communications Magazine* 52.2 (2014), pp. 62–64.

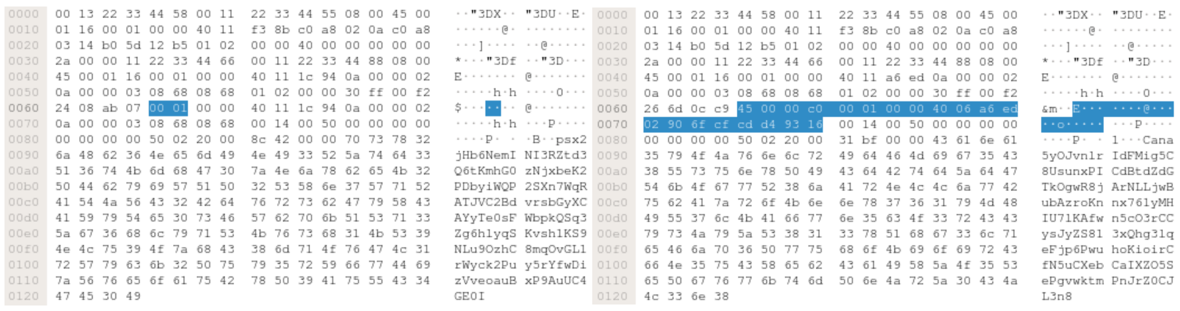
[33] Péter Vörös et al. “T4P4S: A Target-independent Compiler for Protocol-independent Packet Processors”. In: *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR).* IEEE. 2018, pp. 1–8.

[34] Qi Zhang, Jianhui Liu, and Guodong Zhao. “Towards 5G enabled tactile robotic telesurgery”. In: *arXiv preprint arXiv:1803.03586* (2018).



```
No.   Time       Source            Destination       Protocol        Length  Info
    1 0.000000   2.144.111.207     205.212.147.22    GTP <TCP>          292  20 → 80 [SYN…
    2 0.000109   10.0.0.2          10.0.0.3          GTP <0x0001>       292  PPP Padding …
    3 0.000205   10.0.0.2          10.0.0.3          GTP <0x0001>       292  PPP Padding …
    4 0.000359   10.0.0.2          10.0.0.3          GTP <0x0001>       292  PPP Padding …
    5 0.000361   33.35.162.70      14.35.208.248     GTP <TCP>          292  20 → 80 [SYN…
    6 0.000502   101.43.145.194    219.27.187.136    GTP <TCP>          292  20 → 80 [SYN…
    7 0.000557   186.120.108.186   126.33.116.71     GTP <TCP>          292  20 → 80 [SYN…
    8 0.000674   5.29.239.50       14.79.225.195     GTP <TCP>          292  20 → 80 [SYN…
    9 0.000789   17.24.134.1       101.206.105.91    GTP <TCP>          292  20 → 80 [SYN…
   10 0.000911   10.0.0.2          10.0.0.3          GTP <0x0001>       292  PPP Padding …
```

Figure A.1: Packet dump after 50% workload sharing between SmartNIC and T4P4S-T4P4S error detected



```
0000  00 13 22 33 44 58 00 11  22 33 44 55 08 00 45 00   ··"3DX·· "3DU··E·
0010  01 16 00 01 00 00 40 11  f3 8b c0 a8 02 0a c0 a8   ······@· ········
0020  03 14 b0 5d 12 b5 01 02  00 00 40 00 00 00 00 00   ···]···· ··@·····
0030  2a 00 00 11 22 33 44 66  00 11 22 33 44 88 08 00   *···"3Df ··"3D···
0040  45 00 01 16 00 01 00 00  40 11 1c 94 0a 00 00 02   E······· @·······
0050  0a 00 00 03 08 68 08 68  01 02 00 00 30 ff 00 f2   ·····h·h ····0···
0060  24 08 ab 07 00 01 00 00  40 11 1c 94 0a 00 00 02   $······· @·······
0070  0a 00 00 03 08 68 08 68  00 14 00 50 00 00 00 00   ·····h·h ···P····
0080  00 00 00 00 50 02 20 00  8c 42 00 00 70 73 78 32   ····P· · ·B··psx2
0090  6a 48 62 36 4e 65 6d 49  4e 49 33 52 5a 74 64 33   jHb6NemI NI3RZtd3
00a0  51 36 74 4b 6d 68 47 30  7a 4e 6a 78 62 65 4b 32   Q6tKmhG0 zNjxbeK2
00b0  50 44 62 79 69 57 51 50  32 53 58 6e 37 57 71 52   PDbyiWQP 2SXn7WqR
00c0  41 54 4a 56 43 32 42 64  76 72 73 62 47 79 58 43   ATJVC2Bd vrsbGyXC
00d0  41 59 79 54 65 30 73 46  57 62 70 6b 51 53 71 33   AYyTe0sF WbpkQSq3
00e0  5a 67 36 68 6c 79 71 53  4b 76 73 68 31 4b 53 39   Zg6hlyqS Kvsh1KS9
00f0  4e 4c 75 39 4f 7a 68 43  38 6d 71 4f 76 47 4c 31   NLu9OzhC 8mqOvGL1
0100  72 57 79 63 6b 32 50 75  79 35 72 59 66 77 44 69   rWyck2Pu y5rYfwDi
0110  7a 56 76 65 6f 61 75 42  78 50 39 41 75 55 43 34   zVveoauB xP9AuUC4
0120  47 45 30 49                                        GE0I
```

(a) In T4P4S - incorrect inner1_IP header

```
0000  00 13 22 33 44 58 00 11  22 33 44 55 08 00 45 00   ··"3DX·· "3DU··E·
0010  01 16 00 01 00 00 40 11  f3 8b c0 a8 02 0a c0 a8   ······@· ········
0020  03 14 b0 5d 12 b5 01 02  00 00 40 00 00 00 00 00   ···]···· ··@·····
0030  2a 00 00 11 22 33 44 66  00 11 22 33 44 88 08 00   *···"3Df ··"3D···
0040  45 00 01 16 00 01 00 00  40 11 a6 ed 0a 00 00 02   E······· @·······
0050  0a 00 00 03 08 68 08 68  01 02 00 00 30 ff 00 f2   ·····h·h ····0···
0060  26 6d 0c c9 45 00 00 c0  00 01 00 00 40 06 a6 ed   &m··E··· ····@···
0070  02 90 6f cf cd d4 93 16  00 14 00 50 00 00 00 00   ··o····· ···P····
0080  00 00 00 00 50 02 20 00  31 bf 00 00 43 61 6e 61   ····P· · 1···Cana
0090  35 79 4f 4a 76 6e 6c 72  49 64 46 4d 69 67 35 43   5yOJvnlr IdFMig5C
00a0  38 55 73 75 6e 78 50 49  43 64 42 74 64 5a 64 47   8UsunxPI CdBtdZdG
00b0  54 6b 4f 67 77 52 38 6a  41 72 4e 4c 4c 6a 77 42   TkOgwR8j ArNLLjwB
00c0  75 62 41 7a 72 6f 4b 6e  6e 78 37 36 31 79 4d 48   ubAzroKn nx761yMH
00d0  49 55 37 6c 4b 41 66 77  6e 35 63 4f 33 72 43 43   IU7lKAfw n5cO3rCC
00e0  79 73 4a 79 5a 53 38 31  33 78 51 68 67 33 6c 71   ysJyZS81 3xQhg3lq
00f0  65 46 6a 70 36 50 77 75  68 6f 4b 69 6f 69 72 43   eFjp6Pwu hoKioirC
0100  66 4e 35 75 43 58 65 62  43 61 49 58 5a 4f 35 53   fN5uCXeb CaIXZO5S
0110  65 50 67 76 77 6b 74 6d  50 6e 4a 72 5a 30 43 4a   ePgvwktm PnJrZ0CJ
0120  4c 33 6e 38                                        L3n8
```

(b) In SmartNIC - correct headers

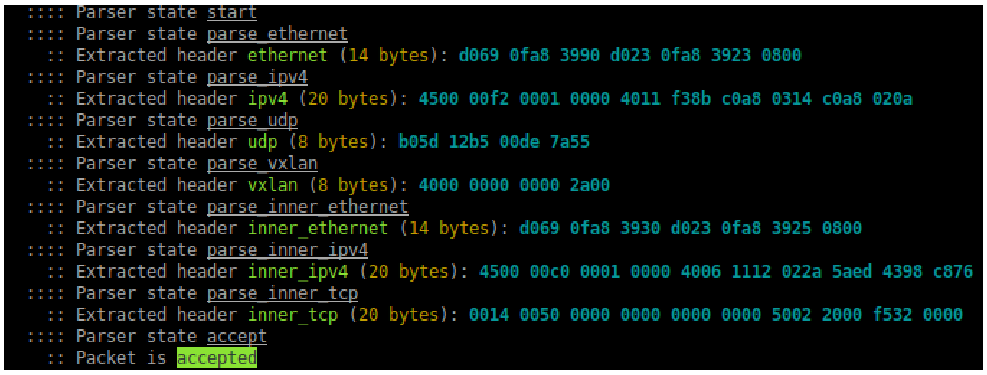
Figure A.2: Raw packet dump after encapsulation, where the header directly following GTP is highlighted, showing inconsistencies between targets

A Appendix

A.1 T4P4S Incorrect Encapsulation

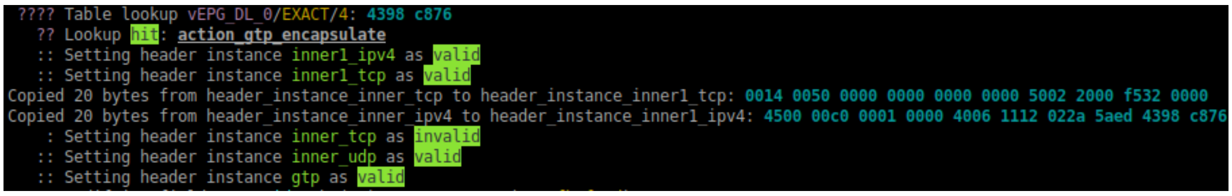
While verifying the pipeline functionality running on Netronome and T4P4S, incorrect headers was detected in packets processed by T4P4S. Downlink traffic was replayed through the hybrid target, with a 50/50 workload split between the sNIC and T4P4S targets. Equivalent pipelines are running on these two targets, and the processed packets coming form these two targets are therefore expected to look the same. Figure A.1 shows a raw traffic dump following this test as explained above; it is immediately apparent that there is a problem with packets assigned to T4P4S for processing.

Figure A.2a shows the raw packet data, after being encapsulated in T4P4S. It is apparent that encapsulation has been performed, since the packets have increased in size equal to



```
 :::: Parser state start
 :::: Parser state parse_ethernet
   :: Extracted header ethernet (14 bytes): d069 0fa8 3990 d023 0fa8 3923 0800
 :::: Parser state parse_ipv4
   :: Extracted header ipv4 (20 bytes): 4500 00f2 0001 0000 4011 f38b c0a8 0314 c0a8 020a
 :::: Parser state parse_udp
   :: Extracted header udp (8 bytes): b05d 12b5 00de 7a55
 :::: Parser state parse_vxlan
   :: Extracted header vxlan (8 bytes): 4000 0000 0000 2a00
 :::: Parser state parse_inner_ethernet
   :: Extracted header inner_ethernet (14 bytes): d069 0fa8 3930 d023 0fa8 3925 0800
 :::: Parser state parse_inner_ipv4
   :: Extracted header inner_ipv4 (20 bytes): 4500 00c0 0001 0000 4006 1112 022a 5aed 4398 c876
 :::: Parser state parse_inner_tcp
   :: Extracted header inner_tcp (20 bytes): 0014 0050 0000 0000 0000 0000 5002 2000 f532 0000
 :::: Parser state accept
   :: Packet is accepted
```

Figure A.3: T4P4S debug output during incorrect encapsulation - parser



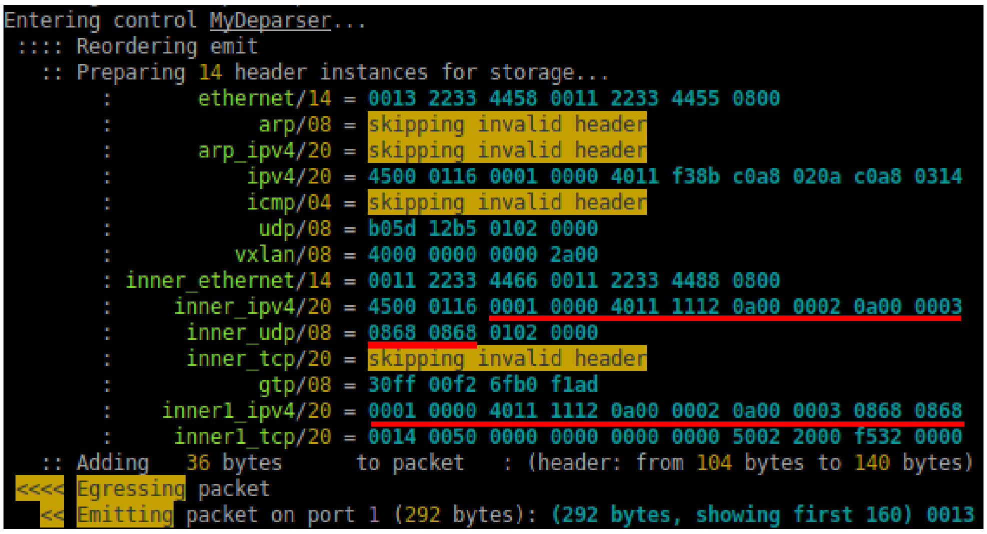
```
 ???? Table lookup vEPG_DL_0/EXACT/4: 4398 c876
   ?? Lookup hit: action_gtp_encapsulate
   :: Setting header instance inner1_ipv4 as valid
   :: Setting header instance inner1_tcp as valid
Copied 20 bytes from header_instance_inner_tcp to header_instance_inner1_tcp: 0014 0050 0000 0000 0000 0000 5002 2000 f532 0000
Copied 20 bytes from header_instance_inner_ipv4 to header_instance_inner1_ipv4: 4500 00c0 0001 0000 4006 1112 022a 5aed 4398 c876
    : Setting header instance inner_tcp as invalid
   :: Setting header instance inner_udp as valid
   :: Setting header instance gtp as valid
```

Figure A.4: T4P4S debug output during incorrect encapsulation - creating encapsulation headers

the inserted encapsulation headers. However, it can also be seen that the *inner1_IP* header directly following the GTP header contains incorrect data. Compare this to a correctly encapsulated packet as seen in figure A.2b, which is assigned for encapsulation in sNIC.

The test is retried, this time with T4P4S debug output enabled, in an attempt to isolate the root cause. As seen in figure A.3, headers are correct as they enter the T4P4S P4 pipeline. Later down in the encapsulation action, values in the *inner1_ip* header are correctly copied from the old *inner_ip* header as seen in figure A.4. T4P4S output shows that no more changes are made to the *inner1_ip* in the ingress pipeline. So far during packet processing, there are no issues.

Further down in the deparsing section of the pipeline, the issue was discovered, as seen in figure A.5; the invalid header data seems to be based in incorrect header storage in T4P4S. The development team behind T4P4S was contacted, and they confirmed that this is a bug in the T4P4S P4 compiler. Shortly after contact, they released a patch for this



```
Entering control MyDeparser...
 :::: Reordering emit
   :: Preparing 14 header instances for storage...
       :        ethernet/14 = 0013 2233 4458 0011 2233 4455 0800
       :             arp/08 = skipping invalid header
       :        arp_ipv4/20 = skipping invalid header
       :            ipv4/20 = 4500 0116 0001 0000 4011 f38b c0a8 020a c0a8 0314
       :            icmp/04 = skipping invalid header
       :             udp/08 = b05d 12b5 0102 0000
       :           vxlan/08 = 4000 0000 0000 2a00
       : inner_ethernet/14 = 0011 2233 4466 0011 2233 4488 0800
       :     inner_ipv4/20 = 4500 0116 0001 0000 4011 1112 0a00 0002 0a00 0003
       :      inner_udp/08 = 0868 0868 0102 0000
       :      inner_tcp/20 = skipping invalid header
       :            gtp/08 = 30ff 00f2 6fb0 f1ad
       :    inner1_ipv4/20 = 0001 0000 4011 1112 0a00 0002 0a00 0003 0868 0868
       :     inner1_tcp/20 = 0014 0050 0000 0000 0000 0000 5002 2000 f532 0000
   :: Adding   36 bytes      to packet   : (header: from 104 bytes to 140 bytes)
 <<<< Egressing packet
   << Emitting packet on port 1 (292 bytes): (292 bytes, showing first 160) 0013
```

Figure A.5: T4P4S debug output during incorrect encapsulation - deparser

bug[5], which was confirmed to fix this issue of incorrect GTP encapsulation in T4P4S.

A.2 Test Case Generating Code

A.2.1 Packet Trace Generation

```python
#Generate vepg packets, return list of generated packets
def genvepgPackets(count, psize, uplink, tsprepend, varyPorts = False, markTarget =
    ↪ False, markerIDs = []):
  pkts = []
  if markTarget == True:
  if count > len(markerIDs):
    print("ERROR! List of markers shorter than number of packets!")
    exit()
  for i in tqdm(range(0, count)):
    pkt = Ether(src = DEFAULT_MACSRC, dst = DEFAULT_MACDST)
    if markTarget:
      pkt /= IP(src = DEFAULT_IPSRC, dst = DEFAULT_IPDST, id = markerIDs[i])
```

[5]This bug was fixed in T4P4S commit 8eea9be53e13ec98f9980cf8c164ddd0df10f271, released on May 5th 2020

```
    else:
      pkt /= IP(src = DEFAULT_IPSRC, dst = DEFAULT_IPDST)
    #Having static ports could hinder RSS loadbalancing
    if varyPorts == True:
      pkt = pkt / UDP(sport = i%60000, dport = DEFAULT_DPORT)
    else:
      pkt = pkt / UDP(sport = DEFAULT_SPORT, dport = DEFAULT_DPORT)
    pkt = pkt / VXLAN(flags = 0x40, vni = 42) / Ether(src = DEFAULT_MACSRC2, dst =
    ↪ DEFAULT_MACDST2)
    #Check if should prepend OSNT TS header
    if tsprepend:
      pkt = Raw('A' * 16) / pkt
    #Uplink includes gtp encapsulation
    if uplink:
      pkt /= IP(src = DEFAULT_IPSRC2, dst = DEFAULT_IPDST2) / \
        UDP(sport = DEFAULT_PORT2, dport = DEFAULT_PORT2) / \
        GPRS_CUSTOM(flags = 0x30, msgtype = 0xff, length = psize - 100 + 10, teid =
    ↪ iptable[i]['teid'])
    pkt /= IP(src = iptable[i]['srcip'], dst = iptable[i]['dstip']) / \
      TCP(sport = 20, dport = 80)
    if len(pkt) > psize:
      print("Error! Packet too short! Current size without payload is: %i!" %len(pkt))
      print("Aborting")
      exit(0)
    #Add random payload to packet, pkt will total psize
    payloadsize = psize - len(pkt)
    pkt /= Raw(RandString(size=payloadsize))
    pkts.append(pkt)
  return pkts
```

A.3 Table Configuration Files

A.3.1 Tofino

```json
{
  "priority": 4000,
  "tableId": 1,
  "deviceId": "device:tofino",
  "isPermanent": "true",
  "appId": "vepg-pipeconf",
  "state": "ADD",
  "treatment": {
    "instructions": [
      {
      "type": "PROTOCOL_INDEPENDENT",
      "subtype": "ACTION",
      "actionId": "SwitchIngress.gtp_encapsulate",
      "actionParams": {
        "teid": "266d0cc9"
      }
    }
    ]
  },
  "selector": {
    "criteria": [
    {
      "type": "IPV4_DST",
      "ip": "205.212.147.22/32"
    }
    ]
  }
},
```

Listing 9: Snippet from Tofino table configuration JSON, showing a single rule for vEPG_DL table

A.3.2 Netronome

```
"ingress::dmac": {
  "rules": [
    {
      "action": {
        "type": "ingress::nop"
      },
      "name": "Mac1_DMAC",
      "match": {
        "ethernet.dstAddr": {
          "value": "d0:69:0f:a8:39:90"
        }
      }
    }
  ],
  "default_rule": {
  "action": {
    "type": "ingress::drop"
  },
  "name": "application_default"
  }
},
```

Listing 10: Snippet from Netronome table configuration JSON, showing configuration for dmac table

```
"ingress::dmac": {
  "rules": [
    {
      "action": {
        "type": "ingress::nop"
      },
      "name": "Mac1_DMAC",
      "match": {
        "ethernet.dstAddr": {
          "value": "d0:69:0f:a8:39:90"
        }
      }
    }
  ],
  "default_rule": {
    "action": {
      "type": "ingress::drop"
    },
    "name": "application_default"
  }
},
```

Listing 11: Snippet from Netronome table configuration JSON, showing configuration for dmac table

```
"ingress::firewall_DL": {
  "default_rule": {
    "action": {
      "type": "ingress::nop"
    },
    "name": "application_default"
  }
},
```

Listing 12: Snippet from Netronome table configuration JSON, showing configuration for firewall_DL table

```
"ingress::firewall_UL": {
  "default_rule": {
    "action": {
      "type": "ingress::nop"
    },
    "name": "application_default"
  }
},
```

Listing 13: Snippet from Netronome table configuration JSON, showing configuration for firewall_UL table

```
"ingress::smac": {
  "rules": [
    {
      "action": {
        "type": "ingress::mac_learn"
      },
      "name": "Mac1_SMAC",
      "match": {
        "ethernet.srcAddr": {
          "value": "d0:23:0f:a8:39:23"
        }
      }
    }
  ],
  "default_rule": {
    "action": {
      "type": "ingress::mac_learn"
    },
    "name": "application_default"
  }
},
```

Listing 14: Snippet from Netronome table configuration JSON, showing configuration for smac table

```
"ingress::vEPG_UL": {
  "rules": [
    {
      "action": {
        "type": "ingress::gtp_decapsulate"
      },
      "name": "UL_IP1",
      "match": {
        "inner_ipv4.dstAddr": {
          "value": "10.0.0.2"
        }
      }
    }
  ],
  "default_rule": {
    "action": {
      "type": "ingress::drop"
    },
    "name": "application_default"
  }
},
```

Listing 15: Snippet from Netronome table configuration JSON, showing configuration for vEPG_UL table

```
"ingress::vEPG_DL": {
  "rules": [
    {
      "action": {
```

```
      "data": {
        "teid": {
          "value": "644680905"
        }
      },
      "type": "ingress::gtp_encapsulate"
    },
    "name": "DL_RULE_0",
    "match": {
      "inner_ipv4.dstAddr": {
        "value": "205.212.147.22"
      }
    }
  },
  {
    "action": {
      "data": {
        "teid": {
          "value": "604547847"
        }
      },
      "type": "ingress::gtp_encapsulate"
    },
    "name": "DL_RULE_1",
    "match": {
      "inner_ipv4.dstAddr": {
        "value": "37.181.194.235"
      }
    }
  }
],
"default_rule": {
  "action": {
      "type": "ingress::tohost"
    },
    "name": "application_default"
  }
}
```

Listing 16: Snippet from Netronome table configuration JSON, showing configuration for vEPG_DL table with just two TEID mappings

A.3.3 T4P4S

```
205.212.147.22 644680905
37.181.194.235 604547847
```

Listing 17: Complete T4P4S vEPG_DL configuration file, for test case with just two TEID mappings

Acronyms

5G Fifth Generation mobile networks. 1, 4, 5, 14

5GC 5G Core Network. 3, 4, 5, 6

ARP Address Resolution Protocol. 9

ASIC Application Specific Integrated Circuit. 7, 8, 11

BSP Board Support Package. 27

CPU Central Processing Unit. 2, 3, 7, 11, 12, 22, 28, 29, 36, 37, 50

CSV Comma-Separated Values. 38

CUPS Control and User Plane Separation. 4, 6

DCGW Data Center Gateway. 7, 35, 36

DPDK Data Plane Development Kit. 2, 12, 13, 15, 16, 17, 18, 27, 28, 29, 30, 37, 40, 41, 43, 50

DPI Deep Packet Inspection. 51

DUT Device Under Test. 36, 39, 41

eNodeB E-UTRAN Node B. 7, 35

EPC Evolved Packet Core. 4, 6, 7

EPG Evolved Packet Gateway. 6

FPGA Field-Programmable Gate Array. 8, 33

GTP GPRS Tunneling Protocol. 1, 2, 3, 6, 7, 8, 12, 15, 16, 17, 18, 21, 22, 25, 31, 32, 38, 39, 45, 46, 50

IOMMU Input–Output Memory Management Unit. 28

IP Internet Protocol. 8, 9, 18, 22, 23, 25, 30, 31, 32, 36, 39, 45

IPG Inter-Packet Gap. 34, 35, 37

LPM Longest Prefix Match. 45

LTE Long-Term Evolution. 4, 5, 7

MAC Medium Access Control. 19, 21

NDR Non Drop Rate. 38

NFP Network Flow Processor. 11, 12, 15, 27

NFV Network Function Virtualization. 1, 2, 3, 6, 8, 14, 40, 44, 51

NIC Network Interface Card. 2, 8, 11, 12, 13, 14, 15, 16, 17, 28, 40, 41, 44, 47

NSH Network Service Header. 18

ONOS Open Network Operating System. 26

OSNT Open Source Network Tester. 25, 27, 33, 34, 35, 36, 37, 38, 39

PCAP Packet Capture. 24, 33, 34, 36, 37

PDR Partial Drop Rate. 38, 41

PF Physical Function. 16

P-GW Packet Data Network Gateway. 5

PMD Poll Mode Driver. 12, 28

PPS Packets Per Second. 35, 36

RAN Radio Access Network. 7

RSS Receive-Side Scaling. 13, 25, 34

RX Receive. 12, 13, 25, 28, 30, 35, 38

SDK Software Development Kit. 27

S-GW Serving Gateway. 5

sNIC SmartNIC. 2, 3, 11, 12, 15, 16, 17, 18, 19, 21, 22, 23, 25, 26, 27, 28, 29, 32, 37, 39, 40, 41, 43, 44, 45, 46, 47, 48, 49, 50, 51, 55, 56

SR-IOV Single-Root Input/Output Virtualization. 12, 16, 17, 27

TEID Tunnel Endpoint Identifier. 8, 16, 17, 22, 23, 24, 25, 26, 30, 31, 32, 40, 43, 44, 45, 46, 49

ToR Top-of-Rack. 7

TX Transmit. 13, 35

UPF User Plane Function. 4, 6, 18, 21

URLLC Ultra-Reliable and Low-Latency Communication. 1, 5, 51

vEPG Virtual Evolved Packet Gateway. 4, 6, 7, 8, 15, 18, 27, 30, 39, 46, 47, 50

VF Virtual Function. 12, 17, 18, 27, 28, 29, 30, 37, 40, 43

VNF Virtual Network Function. 1, 2, 3, 4, 6, 7, 14, 15, 38, 44, 45, 47, 51