P4NFV: An NFV Architecture with Flexible Data Plane Reconfiguration

Mu He, Arsany Basta, Andreas Blenk, Nemanja Deric, Wolfgang Kellerer
Chair of Communication Networks, Technical University of Munich, Germany
{mu.he,arsay.basta,andreas.blenk,nemanja.deric,wolfgang.kellerer}@tum.de

***Abstract*—Current architecture proposals for Network Function Virtualization (NFV) do not integrate hardware-accelerated network function implementations. Recent research studies have shown that pure software-based implementations cannot achieve the needed line rates for todays network services. We propose P4NFV to fill this gap. Making use of an additional abstraction layer, P4NFV is an architecture that can achieve software-based network function implementations as well as handle P4 for programming protocol-independent packet processors. With P4NFV, network operators can still instantiate network functions that are specified in terms of computing and storage hardware, while making use of the performance improvements of P4-enhanced networking hardware. Moreover, in order to take the fast changing nature of todays network services into account, P4NFV integrates mechanisms to reconfigure P4-based network functions at runtime: another missing gap in literature. Based on a proof-of-concept implementation of P4NFV for four network functions, we show promising measurement results. Whereas the network function implementations tailored towards reconfigurations add only marginal overhead, even configuring network functions at runtime does not notably affect network service operations with higher latency or severe packet loss.**

***Index Terms*—Network Architecture, Network Function Virtualization, Programmable Data Plane, Network Function Adaptation.**

I. Introduction

Emerging applications and changing user demands are challenging today's communication networks such as the Internet: applications like augmented reality require low latency, while big data applications introduce new dimensions of network traffic in terms of scale and dynamicity. Such demands stand in stark contrast to the inflexible nature of the legacy infrastructure. Hardware-based routers and middleboxes, which cannot be easily upgraded or reconfigured, hinder the network operators to update their infrastructure. To address these issues, Network Function Virtualization (NFV) has emerged to enable more flexible communication networks [1]. NFV deploys Network Functions (NFs) as software running on commodity servers. It decouples function logic from hardware realization. As a result, operators can flexibly instantiate, configure, migrate and terminate NF according to dynamic system conditions [2]; hence operators can efficiently adapt the infrastructure.

A pure software solution as proposed by recent NFV approaches may also raise concerns. First, the line rate processing target is hard to achieve in a software-based NF, especially for small packets ($\leq$ 128 Bytes) [3]. Second, many NFs, such as Deep Packet Inspection (DPI) and packet en/decryption, are compute-intensive. The appliance of general purpose CPUs, which are not designed specifically for those tasks, is not cost-effective from the techno-economic perspective [4]. Thus, as a next step to address the issues of pure software-based implementations, the combination of software and programmable hardware is a promising research direction [5], [6]. In this regard, P4 has been introduced to program software and hardware networking devices. P4 promises to combine the advantages of both building blocks: better packet processing performance due to hardware-based networking implementations and the flexibility of software-based programmability of network operations.

When compared to pure software implementations (without P4), the nature of P4 promises several advantages in the NFV scenario. First, NFs can be implemented with less amount of code and the NF development phase becomes more efficient [7]. Second, it is possible to simultaneously manage the NFs that operate as software and in hardware. As an example, the same P4 code of a firewall can be loaded in the BMv2 software [8] or in the NetFPGA chip [9]. Moreover, the runtime reconfigurability of P4 assists the flexible deployment of NFs in the face of network dynamics [10]. Therefore, combining P4 with NFV is promising to create a reconfigurable data plane with high management efficiency.

The combination of P4 and NFV, however, is still missing in the literature: there is no work that uses P4 to implement NFs that can operate as software or hardware and can be reconfigured at runtime. In this regard, we propose P4NFV, which abstracts the underlying physical infrastructure as a set of NF nodes running P4 programs with runtime reconfiguration support. As state consistency becomes an issue in case of reconfigurations, P4NFV preserves the consistency of NFs during reconfiguration. The contributions are as follows:

- Proposing an NFV management architecture for P4-enhanced data planes.
- Presenting two approaches to reconfigure P4-based network functionality at runtime, and analyzing the approaches' trade-offs.
- Implementation of a P4NFV prototype and evaluation of NFs' performance during reconfiguration.

The remainder of this paper is structured as follows. Section II provides the background and the related work of data plane programmability with P4, NFV management architecture, and function chaining-related network operations like optimization. Section III introduces the design of the P4NFV architecture. Next, Section IV outlines the details of our prototype implementation, highlighting the two data plane reconfiguration approaches. Section V provides the evaluation setup and results. Section VI concludes the paper.

II. Background and Related Work

This section briefly overviews data plane programmability with P4, NFV management architectures, and data plane reconfiguration approaches.

A. Data Plane Programmability with P4

Data plane programmability has drawn much attention in the networking community. An merging approach to program the data plane is to express the forwarding behavior with domain specific languages, such as P4, and then to compile it to a program/configuration that can be loaded in a target.

1) The P4 Language: Bosshart et al. [10] first proposed the P4 programming language. All the basic building blocks of packet processing, i.e., header parsing, actions based on header matching and re-encapsulation, can be described with P4. The header parsing is implemented as a state machine to identify a sequence of headers and to extract them into metadata for later processing. Following the Match+Action paradigm, the match tables, associated with the user-defined actions, define the pipeline of packet processing, and the tables are populated via a control plane interface. Registers, meters and counters can be declared in order to provide stateful implementations.

2) The P4 Targets: P4 programs can be executed in various packet processing entities, i.e., *targets*, from switching software to hardware devices. BMv2 [8] was the first software switch prototype (the so called Simple Switch target). Another software implementation is PISCES [7], which is derived from Open vSwitch [11]. For hardware, SmartNIC [12] leverages the programmability of the Network Flow Processor (NFP), while P4FPGA [13] and P4-to-VHDL [14] enable P4 program execution on FPGAs.

During the compilation of P4, the pipeline is translated into a Table Dependency Graph (TDG) [10] and thereafter mapped into the resources of the specific target. For different targets, the resource mapping can be different: match tables can be mapped to RAM in software switch [7], [8] and TCAM in hardware reconfigurable chips [13], [14].

A software target is assumed to provide the greatest flexibility in building up the Match+Action pipeline; however, packets are processed one after the other, which can lead to low resource utilization. On the contrary, for an NFP, the corresponding P4 compiler can explore the parallelism of stages in a pipeline. The parallelism leads to higher utilization, and therefore line-rate packet forwarding can be achieved [15]. Besides, specific logic transistors enable even higher packet processing efficiency, compared with general purpose CPUs [16].

Due to the increasing variety of P4 targets, performance analyses are needed to identify and understand potential trade-offs. Such understanding makes it possible to select the proper platform for a specific use case. Current performance evaluations (such as [7], [14], [17]) focus merely on static operations, e.g., packet processing latency and throughput. It can be noted that different targets introduce various implementation aspects, which should be considered during implementation and deployment of NFs. Hence, P4NFV introduces an abstraction layer to take care of the various target-specific details.

B. NFV Management Architectures

A variety of NFV architectures have been proposed to tackle various aspects of resource management in NFV [18]–[21], with focus on, e.g., automation of NF provisioning, NF latency reduction, dynamic resource scheduling, and dynamic service function chaining.

Because of the complexity of software-based network construction, vConductor [18] suggests to automate the NF provisioning procedure, while considering various resource scheduling and infrastructure fault isolation. To reduce latency and increase resource usage efficiency, NetFATE [19] deploys NFs not only in high-performance data center servers, but also in the nodes that are close to end users. NFVnice [20] dynamically schedules resources allocated to service chains and thus enables fair share of CPU to NFs. With SDN support, [21] orchestrates and chains NFs in a data center and demonstrates the high dynamism and flexibility of function chaining compared with legacy hardware architecture. The proposed architectures, however, assume that NFs are implemented as general software, which is not applicable to a data plane for P4 composed of software and hardware programmable devices.

C. Data Plane Reconfiguration Approaches

In this section, we report on work that targets at reconfigurations enabled with and without P4.

1) ***Reconfiguration without P4:*** Several publications [22]–[24] tackle the problem of how to reconfigure the data plane with minimal disruption. The Split/Merge approach [22] considers the dynamic scaling of NFs. A hypervisor abstracts the states of NFs and manages their redistribution upon creating or destroying NF replicas. OpenNF [23] is a control plane architecture that can manage both NF state and networking forwarding state. Special APIs and a combination of events and forwarding updates can redistribute the packet processing across a group of NFs. Zave et al. propose Dysco protocol [24] to enable dynamic service chaining. When the sequence of NFs changes, the protocol reconfigures the data plane packets of the corresponding TCP session with small disruption.

2) ***Reconfiguration leveraging P4:*** The reconfiguration capability of P4 has been leveraged to achieve data plane virtualization, e.g., HyPer4 [25] and HyperV [26]. Like in other virtualization scenarios, the target is to enable the sharing of networking resources between multiple tenants (the ones that receive virtual resources, i.e., a partition of the physical networking resources). Both approaches introduce a hypervisor that enables multiple P4 programs to run isolated on the same packet processing entity. Upon initialization, each processing entity is configured with all necessary P4 programs. A table is used to dispatch the tenant network traffics between the P4 programs of the tenants. By updating certain table entries, the hypervisor can even turn on/off the programs at runtime. We leverage a similar approach to accommodate multiple P4 programs simultaneously on one network processing entity. In contrast to the virtualization approaches, P4NFV focuses on the capability to reconfigure the processing pipelines, i.e., to dynamically steer network traffics between P4 programs running on the same networking entity.

To the best of our knowledge, there is no work that combines NFV and P4 for heterogeneous resources for dynamic use: the NFs are implemented with P4, deployed on various targets, and reconfigured on the fly.

III. Architecture Design

In this section, we first enumerate the challenges of an NFV architecture which leverages data plane reconfigurability. Thereafter, we propose P4NFV, which can manage NFs implemented with P4. Finally, we show that P4NFV complies with the guideline NFV architecture proposed by the ETSI organization [27]. For P4NFV, we assume a scenario where a network provider deploys NFs or whole function chains over time to process network traffic of different network services.

A. Design Goals

To support network dynamics, an NFV management architecture should consider the following three aspects.

1) ***Abstraction****:* In order to balance the trade-off among performance, investment and revenue, infrastructure providers should be able to deploy heterogeneous resources, i.e., commodity server and hardware equipment, for NF provisioning. To ease the operation of the heterogeneous resources, abstraction should hide target-specific implementation details, APIs, and performance trade-offs among heterogeneous data plane platforms. Abstracting the infrastructure simplifies the procedure of managing both software and hardware resources. Realizing abstraction via an additional layer can make it possible that infrastructure providers offer various NFs, from lite ones, such as packet forwarder, to advanced ones that are more compute-intensive, such as packet en/decryption.

2) ***Flexibility****:* The architecture should be able to cope dynamics such as changing network traffic conditions or changes in terms of NF's requirements like for emerging service cases. For instance, during the operation of an NF, the requirements from the NF in terms of Quality of Service (QoS) and/or reliability and resilience may alter. The above dynamics should be handled through proper NF management schemes, including feature upgrade, instance migration, parameter adaptation, etc. In other words, it should be possible to instantiate, (re-)configure, (re-)located and upgraded each deployed NF in a flexible manner, with minimum interruption of operation [1].

3) ***Consistency****:* Beside abstraction and flexibility, a holistic architecture design should also integrate consistency. Consistency aims at two aspects: (1) consistency during operation without any adaption and (2) consistency when actually adapting existing NF deployments. First, during operation,

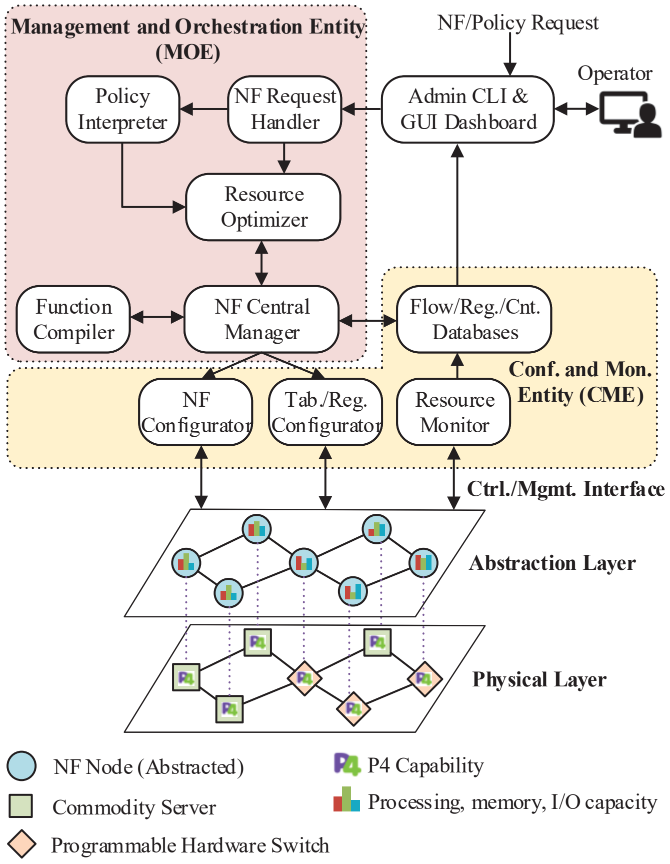


Fig. 1. Illustration of P4NFV architecture design. The physical layer consists of different P4-enabled entities which are abstracted in the abstraction layer for the ease of management.

performance consistency ensures that all abstracted resources provide the capacities as they claim without any unpredictable performance behavior. This might be particularly challenging when facing heterogeneous P4-enabled platforms with performance tradeoffs. Second, when adapting the NF deployments, e.g., when migrating an NF from a hardware to a software target, the performance guarantees should maintain. Moreover, logical consistency should always be preserved, even in case of reconfigurations; e.g., a stateful NF such as a L4 firewalls should not let malicious traffic during reconfigurations.

B. P4NFV Architecture

This section introduces the P4NFV architecture. It overviews its components and clarifies how P4NFV realizes the design goals as mentioned in the last section. Fig. 1 illustrates the architecture of P4NFV. It is logically composed of five components: the Physical Layer, the Abstraction Layer, the Control and Management Interface (CMI), the Configuration and Monitor Entity (CME), and the Management and Orchestration Entity (MOE).

1) ***Physical Layer****:* The physical layer consists of various types of packet processing entities (targets) that can be programmed with P4. As shown in Fig. 1, green squares represent commodity servers that can host software targets, and orange squares represent programmable hardware targets. Note that the physical layer can be extended to incorporate other entities which are not programmable. For instance, commodity routers with configurable forwarding information bases can also be controlled in the architecture. Such hybrid infrastructure can benefit from both NF deployment flexibility [4] and CAPEX/OPEX cost saving [19].

*2) **Abstraction Layer (NF Nodes)**:* This layer abstracts the physical resources. All the *abstracted* processing entities can implement the Match+Action pipelines as demanded by P4. Specifically, each entity is abstracted as an NF node, which is equipped with processing, memory and I/O resources and is able to host multiple NFs. Through the abstraction, various targets are modeled with their own performance characteristics, e.g., whether they can process packets in parallel or note. The performance models assist the network operator to decide the best target for a particular NF requirement. To preserve data plane consistency, the resources are monitored by the upper components to alleviate, e.g., overload situations, which may cause data plane performance degradation.

*3) **Control and Mangement Interface (CMI)**:* This interface is the logical communication channel between *CME* (introduced later) and the underlying NF nodes. For different targets, the actual implementation of *CMI* can be different; however, the distinction is complete transparent to the above components in P4NFV. All the operations, e.g., push compiled P4 programs to NF nodes, populate match tables, fetch counter values and read/write registers, pass through this interface.

*4) **Configuration and Monitor Entity (CME)**:* The components of the Configuration and Monitor Entity (*CME*) are the *Configurator*s, the *Resource Monitor* and the *Databases*. Based on the configurations received from *MOE*, the *Configurator*s take the responsibility of implementation of the NFs in the respective physical NF nodes.

The *Resource Monitor* collects the statistics from the *Abstraction Layer* periodically and notifies *MOE* in an event-based fashion, i.e., whenever any performance indicator, e.g., the load balancing factor and physical link utilization, violates a predefined threshold. Besides, the *Resource Monitor* also collects the values of registers and counters and keeps such state information in the *Databases*. With the help of registers, flow information can be stored, such as header fingerprint, average arrival rate, and forwarding state. The *Database* approach has a clear advantage: it can help to maintain the global consistency of various NF instances during reconfiguration.

*5) **NF Management and Orchestration Entity (MOE)**:* As the central component of P4NFV's architecture, it consists of the *NF Request Handler*, the *Resource Optimizer*, the *Policy Interpreter*, the *NF Central Manager*, and the *Function Compiler*. The network operator interacts with *MOE* through the *Admin CLI & GUI Dashboard* module offering different management operation possibilities: e.g., configuring global policies or checking the resource usage of nodes.

MOE automates the whole process of initiating, coordinating and managing NFs. The *NF Request Handler* listens to new NF requests, as well as policy updates of existing NFs. Together with the *NF Request Handler* and the *Policy Interpreter*, the *Resource Optimizer* implements the requests with optimized configurations, including the P4 programs.

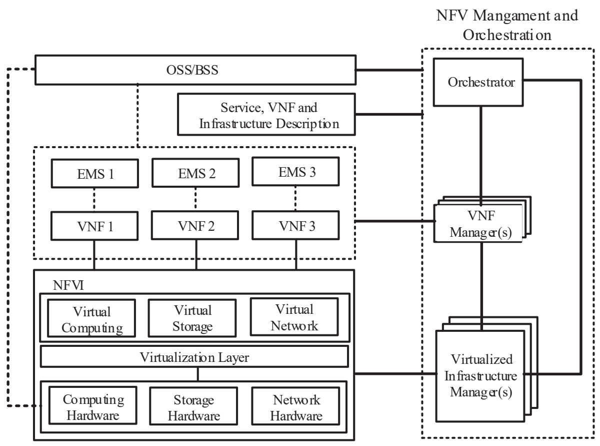


Fig. 2. ETSI NFV reference architecture framework [27]. P4NFV realizes the ETSI architecture, where in addition, physical resources are also P4-enabled.

The configurations are directed to the *NF Central Manager*, and then passed to *CME*, which in turn implements them in the NF nodes. In the meantime, the *NF Central Manager* listens to the data plane status via *CME* and re-optimizes the NF deployment and configuration. The re-optimization can be triggered either by the operator manually, or by the *NF Central Manager* itself. The *Function Compiler* is a set of compilers (e.g., p4c for BMv2 and SDNet for NetFPGA) and is able to compile P4 program for different targets.

C. Mapping to the ETSI Architecture Framework

In this section, we briefly discuss whether P4NFV is compliant with the proposed ETSI NFV architecture. ETSI (European Telecommunication Standards Institute) proposed a guideline NFV reference architecture framework in [27] (shown in Fig. 2). The framework defines the functional blocks and the reference points needed to support the infrastructure services in the operator's network. Within NFV, the infrastructure services are referred to as the network services, which are provided by the NFs. To show the compliance, we map the components of P4NFV to blocks in the framework.

In the reference architecture, the Virtualized Infrastructure Manager (VIM) controls the virtualization process and exposes the NFVI to the other modules. In P4NFV, the *Resouce Monitor* and the *Resource Optimizer* take over this part, and the abstraction layer corresponds to the NFVI.

The intermediate level in the reference architecture consists of various VNFs and Element Management Systems (EMSs). Each VNF is an implementation of a network application running atop the NFVI resources. The VNFs' instantiation and termination are controlled by the VNF Manager(s), which is represented by the *NF Central Manager* and *Function Compiler*. During a VNF's lifetime, its management, such as fault recovery, performance monitoring and accounting [27],

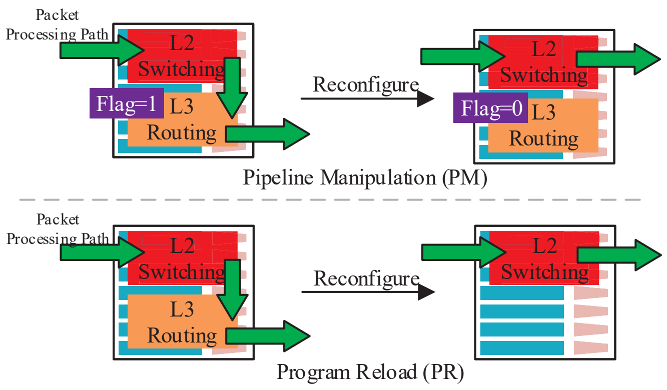


Fig. 3. Illustration of the two proposed data plane reconfiguration approaches. The green arrows represent the path of packet processing. After reconfiguration, only L2 switching NF remains in the NF node.

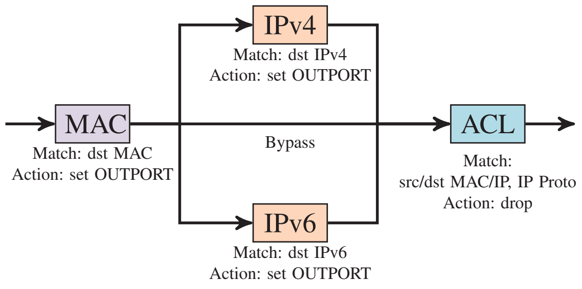


Fig. 4. Packet processing pipeline of a switch.

are handled by its corresponding EMS, which corresponds to the *NF Configurator*, *Table/Register Configuration* and *Resource Monitor* in P4NFV.

The NFV Orchestrator at the top level realizes NF requests by coordinating other modules in the reference architecture. It is represented as the combination of the *NF Request Handler*, *Policy Interpreter* and *Resource Optimizer*.

IV. P4NFV Prototype Implementation Targeting Reconfiguration

Two challenges appear during the prototype implementation: (1) how to reconfigure the data plane during runtime, and (2) how to reduce the impact of reconfigurations on packet processing. We propose two solutions called `Pipeline Manipulation (PM)` and `Program Reload (PR)`.

A. P4NFV Runtime Reconfiguration Mechanisms

Fig. 3 demonstrates the two reconfiguration approaches.

*1) **Pipeline Manipulation (`PM`)**:* In a nutshell, after the P4 program is loaded in the target, we leverage binary register values to manipulate the packet processing Match+Action pipeline at runtime. The pipeline is described by the TDG, which shows the dependency between different tables.

Fig. 4 shows the pipeline of a switch capable of performing L2 and L3 forwarding. MAC/IP forwarding and ACL are considered as different NFs, each guarded with a binary register value to indicate its existence. Upon initialization, all functions are enabled. After we change the register value of MAC forwarding from 1 to 0, the packets will bypass it and jump to the IP forwarding NFs. Similarly, we can disable IP forwarding plus ACL and only keep MAC forwarding.

*2) **Program Reload (`PR`)**:* It provides more flexibility to reconfigure the functionality of the NFs, in comparison with `PM`. Every time a node needs to be reconfigured, the *target* will be loaded with a new compiled P4 program (including parser, processing actions and deparser), followed by populating the new match table entries. Note that because of the implementation limits, some targets, especially hardware ones, may need to stop packet processing completely when a new configuration is being loaded, which causes service interruption.

*3) **Trade-offs between `PM` and `PR`**:* For `PM`, all NF nodes run the same P4 program which includes all NF implementations. The *MOE* only needs to update the registers. However, `PM` demands additional Match+Action resources, which limits the total number of NFs that can be implemented at the same time. Furthermore, since registers can only be applied inside the pipeline, it is not capable of reconfiguring parser and deparser. When it comes to `PR`, different versions of P4 programs are pushed to different NF nodes, which could decrease reliability and increase management cost. Furthermore, `PR` may lead to service interruption for some targets. The advantages of `PR` are (1) that it demands less Match+Action resources, and (2) that it can reduce packet processing latency with proper pipeline compilation (i.e., merging different tables).

B. State Management

During data plane reconfiguration, the states (e.g., register values) that reside in the NFs need to be consistently updated, otherwise the newly arrived packets will experience improper processing. Consider the migration of an IDS (Intrusion Detection System), where packet drops happen during the migration. The new IDS instance may process an incomplete fingerprint of a malicious flow as packets haven been dropped; hence, the IDS may fail to report the attack [28].

There are two alternatives to preserve the states during reconfiguration. First, the states are transferred directly in the data plane together with the live traffic [29]. In this case, NF nodes generate data plane packets with payloads of state information. Second, a central entity collects the states from the data plane and redistributes them. P4NFV applies the second option. The *Resource Monitor* uses *CMI* to collect the states and stores them in the database. Notably, the introduction of *MOE* and *CME* induces latency overhead for state updates. In order to reduce such latency, P4NFV periodically fetches the states associated with the NF and only redistributes the states that are involved in the reconfiguration.

Fig. 5 illustrates the NF migration with `PR`. Four types of messages are involved, which are predefined for the BMv2 target [8]. The *load_new_config_file* indicates that a new data plane will be configured on the target. A series of *table_adds* populate the table entries. State migration consists of a series of *register_read/writes* which copy the register values directly from the source to target NF node. Finally, the *swap_configs* signals the moment when the new data plane would take effect

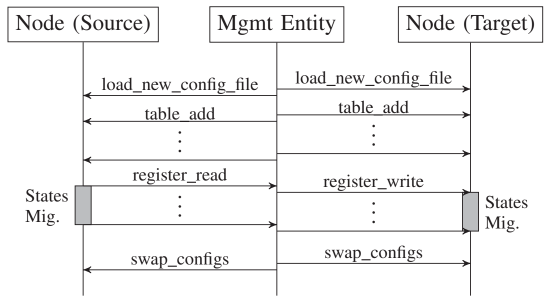


Fig. 5. The message exchange and states of NF (on/off) during migration with PR from an initial to a target NF node.

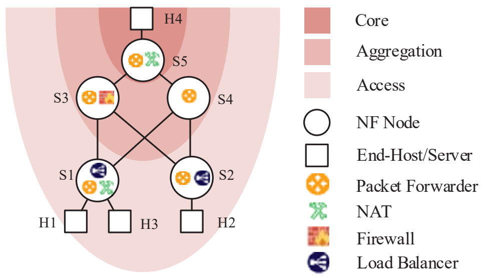


Fig. 6. Topology setup of the use case study with five NF nodes and four types of NFs.

on the target. The procedure of PM is similar to that of PR, but a bit simpler: a series of *table_add*s configure the table entries, and then *write_register* messages turn on and off the NF at the respective NF nodes. Because of packet buffering in BMv2, we do not lose any packet when we switch to a new configuration or change register values in a pipeline.

V. Performance Evaluation

This section first elaborates on P4NFV in a realistic scenario. Afterwards, we take intensive measurements to report on the general performance of our P4NFV implementation in terms of CPU usage and latency. Then we evaluate our two reconfiguration approaches PM and PR.

A. Example Realization of P4NFV for Network Edges

Recently, strict latency requirement and high network traffic overhead in the core drive the infrastructure providers to virtualize and locate NF nodes at the edge of the network infrastructure [19]. Such virtualization reduces the investment and expedites the infrastructure deployment process [19], [30]. Moreover, locating NFs in the proximity of the end-users also contributes to smaller end-to-end latency, which is critical for emerging applications like autonomous driving.

However, the NF nodes deployed at the edge are still limited with resources and thus prone to overload in the face of highly dynamic traffic from end-users. There are two opportunities of improvement introduced by P4NFV: first, P4 itself promises a better and more efficient hardware utilization [16]; second, P4NFV introduces data plane reconfiguration mechanisms that relocate the NF instances if possible.

We implemented four types of NFs for the prototype demonstration: a *Packet Forwarder*, a *NAT*, a *Firewall* and a *Load Balancer*. The *Packet Forwarder* forwards the packets based on destination MAC or IP addresses. It also modifies the source and destination MAC addresses for each packet. Its forwarding rules, stored as table entries, are populated with the *CMI* upon the startup of the network and remain static. The *NAT* is capable of translating private and public IP addresses. The end-users and the core data centers typically have private network addresses, whereas the network entities in between use public network addresses [19]. The *Firewall* detects and blocks malicious flows that originate from the end-users. It works in the stateful manner: other than blocking flows based on static rules, it calculates fingerprints of flows and blocks the ones if the fingerprints violate the predefined policy. In our case, the fingerprint of a flow is its packet inter-arrival rate. The Firewall drops packets of flows whose fingerprints are higher than a threshold.

B. Evaluation Setup and Procedure

We adopt a three-tier topology as presented in [31], which is depicted in Fig. 6. We differentiate three networking domains: access, aggregation, and core. The dark red color represents the core domain, the lighter color the aggregation domain, and the light red color the access domain. Circles labeled from S1-S5 denote the physical nodes hosting the P4-based network functions. Square nodes represent end-hosts and servers in the access and the core domain. The two nodes S1 and S2 in the access region are gateways that connect the end-user nodes H1-H3. Each gateway node is linked to the two aggregation nodes S3 and S4. The aggregation node S5 accumulates traffic from end-users and forwards it to the core node, where H4 represents the server that the end-users want to connect to. For our proof-of-concept implementation, we apply the Simple Switch target [8] to host the NFs and build up the topology with Mininet. For the implementation of *CMI*, we apply BMv2 CLI that comes with the Simple Switch target. We use the D-ITG [32] traffic generator to create network traffic. The evaluations are executed in an environment with Ubuntu 16.04, an Intel Xeon E3-1275v5 CPU of 3.6GHz, and 32 GB of RAM. For each evaluation scenario, we repeat 30 runs to gain statistical confidence.

C. General Performance Evaluation

In the first setting without reconfiguration, UDP packets are sent from H1 to H4. We analyze the impact of different packet rates per second (pps) and payload sizes (Bytes). For our investigations of implementation details, we use one Packer Forwarder NF, which locates on node S1.

1) ***Packet Rate vs. CPU Usage:*** Fig. 7a shows boxplots of the software switch's CPU utilization in percentage over increasing packet rates. The different shapes in the boxplots indicate the corresponding mean values. The CPU utilization increases with the packet rate and the maximum mean value is around 19 %. Whereas the CPU utilization increases with the packet rate, the different payload sizes (50, 500, and 1000) only pose a marginal impact on the CPU utilization. The observation of rate-dependent CPU utilizations motivate for the use of the reconfiguration mechanisms of P4NFV.

*2) **Pipeline vs. CPU Usage**:* The implementation of `PM` can be realized with multiple tables. We now investigate whether the the number of tables can impact the performance of NFs. We use the *Packet Forwarder* located on node S3. We have two alternatives for the implementation. All actions are either implemented in three tables (one table per step: decide output port, update source MAC, update destination MAC) or in one aggregated table. To elaborate on the overhead of additional tables (can be the case for the `PM`), we analyze also an implementation with two more dummy tables mimicking the behavior of an ACL filtering function. Fig. 7b shows the CPU utilization. For the same packet rate, more tables in the pipeline add a CPU utilization overhead, which is notable in case of packet rates from 1300 to 2100.

*3) **Pipeline vs. Latency**:* As for the latency, Fig. 7c reports nearly 20% higher average processing latency when we have five tables instead of one. The results confirm that an NF implementation with more tables induces higher resource utilization and processing latency. The latency and the previous CPU usage results demonstrate the potential overhead of `PM`, whereas with `PR`, multiple tables of different NFs can be compressed in favor of lower resource utilization and latency.

We also observe an interesting phenomenon that the latency decreases when the packet rate is higher. This happens because of the thread-based implementation of the Simple Switch target. In case of higher packet rates, the threads fall less asleep, and it takes less time until the threads wake up to process the packets, which contributes to shorter latency.

D. Stateless NF Migration

We choose the *NAT* as a representative for reconfiguration of a stateless NF. We send UDP traffic from H1 to H4, following the path S1-S3-S5. For each run, we first instantiate the NAT on S1 after 5 seconds and then migrate the NAT to S3 after another 5 seconds. For `PM`, we enable/disable the NAT tables through updating the binary value in the register, whereas for `PR`, we load different P4 programs with/without the NAT implementation, followed by populating the tables accordingly. We analyze two packet rates 1000 pps and 3000 pps for three payload sizes 50, 500, and 1000 Bytes. The plots show the mean values and the 95% confidence intervals of 30 runs.

*1) **Impact on Functionality**:* We first examine the NF's functionality during migration. All UDP packets successfully reach the destination. After dumping all the packets and checking their source IP addresses, we confirm that the IP addresses are modified correctly in all scenarios, meaning that no service disruption happens during NF migration. The BMv2 switch can start working with the new intended packet processing pipeline immediately after the new configuration, e.g., the register values or P4 code, is set via the *CMI*. However, this observation applies mainly to P4 software targets; for hardware targets, additional mechanisms such as buffering may be required to ensure minimal service disruption, i.e., latency increase and potential packet drops.

*2) **Impact on Latency**:* We measure the packet transmission latency from the source to the destination, which reflects the processing time of the NFs along the forwarding path. The results are reported in Fig. 8. In general, the difference between different UDP payload sizes is not significant. For the packet rate of 1k, the difference between `PM` and `PR` is marginal. When the packet rate increases to 3k, peaks show up in the curves when reconfigurations happen. This overhead (more obvious for `PR`) comes from longer queuing delays in the NF.

E. Stateful NF Migration

The stateful firewall NF drops UDP flows that originate from a source IP having a sending rate higher than a threshold. The sending rates are stored in the registers, indexed by hash values that are calculated from packet header 3-tuples (src. IP, dst. IP, and IP protocol number). They indicate whether a flow has to be blocked or not. The stateless load balancer NFs are placed in S1 and S2. They randomly forward packets to balance the load on both links. When performing migration, the *MOE* needs to configure the initial and target NF node in order to preserve the state consistency, e.g., keep dropping the packets of the blocked flows. *MOE* reads the register entries from source NF node S1 and then writes to the target S5.

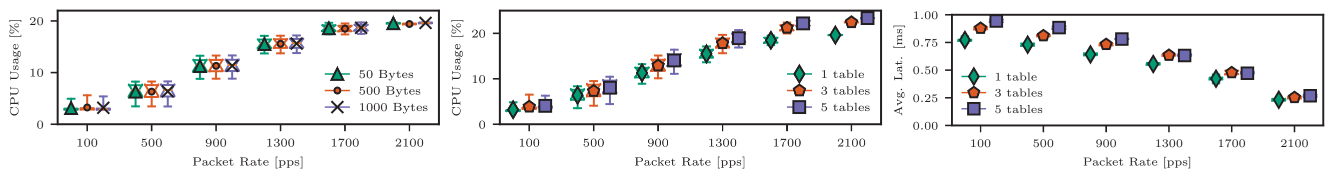
In order to emulate traffic that will be forwarded as well as blocked by the firewall, we create two UDP data sources. The host H1 sends UPD traffic to H4 with a high packet rate — the traffic should be blocked by the firewall. For the concurrent non-blocking traffic, H2 sends 1000 UDP packets per second to H4 with 50 Byte payload. For each run, we initiate the firewall on S1 and then start the traffic generation. The UDP packets from H1 should be blocked, whereas the ones from H2 should always reach the destination. Thereafter, the firewall is migrated at time 5s from S1 to S5. We record the forwarding latencies of all packets that belong to the concurrent flows.

*1) **Impact on Functionality**:* We confirm that no firewall service disruption happens during migration for any scenario, as no packets of the blocked flow reaches the destination server. The state that indicates the blocking of H1's flow is copied from S1 to S5 before the firewall is actually migrated. Thus P4NFV is able to preserve the state information of the firewall during its migration.

*2) **Impact on Latency**:* We do not observe any packet loss of the concurrent traffic. However, as illustrated in Fig. 9, the migration indeed poses an impact on the forwarding latency. In general, `PR` introduces slightly higher latency (0.6ms) than `PM` (0.5ms), and `PR`'s performance degradation lasts 0.3s longer than `PR`. Because of more packets buffering, the maximal delay during firewall migration of 3k pps can be two times of 1k pps. Such delay can be alleviated by applying P4 targets that support parallel packet processing.

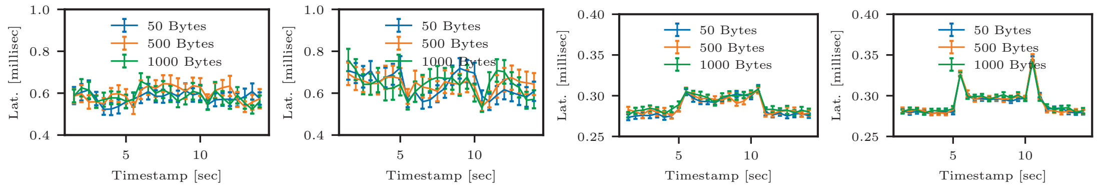
F. Comparison With a Legacy NFV Solution

Following the legacy NFV solution, we implement pure software-based NFs running inside VMs. We deploy the VMs



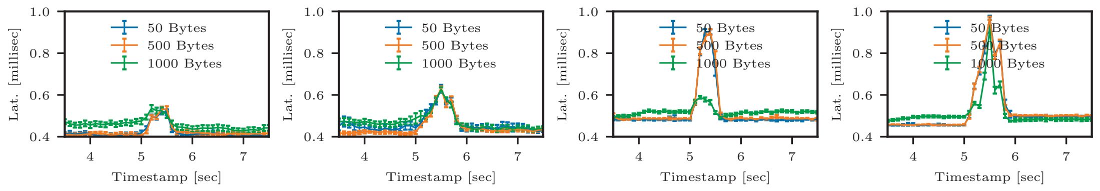
(a) CPU Usage Comparison - Traffic (b) CPU Usage Comparison - Tables (c) Latency Comparison - Tables

Fig. 7. (a) shows the relation between the CPU usage of different payload sizes and packet rates. (b) shows the relation between the CPU usage of different packet rates and the number of table matches in the pipeline (payload size 50 Bytes). (c) shows the relation between the average latency of different packet rates and the number of table matches in the pipeline (payload size 50 Bytes).



(a) PM - 1k pps (b) PR - 1k pps (c) PM - 3k pps (d) PR - 3k pps

Fig. 8. Impact of NAT (stateless NF) migration on the packet forwarding latency, comparing two reconfiguration approaches and different packet rate. The NAT is instantiated at time 5s, and then migrated at time 10s.



(a) PM - 1k pps (b) PR - 1k pps (c) PM - 3k pps (d) PR - 3k pps

Fig. 9. Impact of firewall (stateful NF) migration on the concurrent traffic, comparing two reconfiguration approaches and different packet rate. The migration is triggered at time 5s.

in an OpenStack cloud and evaluate the performance during VM migrations. We instantiate three VMs which act as the traffic source/sink and the packet processing entity respectively. We implement the NAT and the stateful firewall in Python with the Scapy library [33]. The NF migration makes use of the VM live-migration option of OpenStack [34].

For the NAT scenario, a flow with 500 pps is generated for 15s (7500 to be transmitted in total). On average 108.43 packets are lost during the migration of the NAT, which corresponds to a service disruption of 0.217s. For the firewall scenario, two flows with 500 pps are generated for 15s. Because the state is stored in the firewall VM, there is no need to coordinate the state migration. Also for the VM-based NF setting, no packets of the blocked flow reach the sink. However, the non-blocked flow experiences a similar packet loss as in the NAT scenario: on average 120.53 packets are lost, which corresponds to a service disruption of 0.241s.

In contrast to the legacy solution, P4NFV can migrate both NFs without any service interruption. For the performance, we observe only a short latency increase during migration.

VI. Conclusion

Leveraging the data plane programmability of P4, we propose an NFV architecture that is able to use hybrid infrastructure resources and reconfigure the network functionalities in the field. The architecture preserves the consistency of stateful NFs during reconfigurations. Two approaches to achieve runtime reconfiguration are proposed with the consideration of network state management. Based on a prototype implementation, we evaluate various performance indicators of the architecture. Static performance evaluations motivate the necessity of NF relocation in face of dynamic traffic, as well as the careful design of the pipeline structure. As a highlight, we provide comprehensive evaluations of the data plane performance during runtime reconfiguration. In comparison with a conventional VM solution, P4NFV ensures the liveness of functions and acceptable performance degradation when functions are migrated. As our reconfigurations introduce packet loss, we believe that reconfiguration mechanisms avoiding losses are interesting future work.

Acknowledgment

This work has been funded by ERC FlexNets Project (grant No 647158), the German BMBF SENDATE-PLANETS Project (ID C2015/3-1), and the DFG ModaNet Project (grant No KE 1863/8-1). The authors alone are responsible for the content of the paper.

REFERENCES

[1] W. Kellerer, A. Basta, P. Babarczi, A. Blenk, M. He, M. Kluegel, and A. Martinez-Alba, "How to measure network flexibility? a proposal for evaluating softwarized networks," *IEEE Communications Magazine*, 2018.

[2] A. Greenhalgh, F. Huici, M. Hoerdt, P. Papadimitriou, M. Handley, and L. Mathy, "Flow processing and the rise of commodity network hardware," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 2, pp. 20–26, 2009.

[3] Z. Niu, H. Xu, L. Liu, Y. Tian, P. Wang, and Z. Li, "Unveiling performance of NFV software dataplanes," in *Proceedings of the 2nd Workshop on Cloud-Assisted Networking*. ACM, 2017, pp. 13–18.

[4] Z. Bronstein, E. Roch, J. Xia, and A. Molkho, "Uniform handling and abstraction of NFV hardware accelerators," *IEEE Network*, vol. 29, no. 3, pp. 22–29, 2015.

[5] N. Zilberman, P. M. Watts, C. Rotsos, and A. W. Moore, "Reconfigurable network systems and software-defined networking," *Proceedings of the IEEE*, vol. 103, no. 7, pp. 1102–1124, 2015.

[6] H. Moens and F. De Turck, "Customizable function chains: Managing service chain variability in hybrid NFV networks," *IEEE Transactions on Network and Service Management*, vol. 13, no. 4, pp. 711–724, 2016.

[7] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford, "PISCES: A programmable, protocol-independent software switch," in *Proceedings of ACM SIGCOMM*. ACM, 2016, pp. 525–538.

[8] "P4 behavioral-model," https://github.com/p4lang/behavioral-model/, accessed: 2018-02-20.

[9] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo, "NetFPGA–an open platform for gigabit-rate network switching and routing," in *Proceedings of IEEE International Conference on Microelectronic Systems Education*. IEEE, 2007, pp. 160–161.

[10] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[11] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, and P. Shelar, "The Design and Implementation of Open vSwitch," in *Proceedings of NSDI*, vol. 15, 2015, pp. 117–130.

[12] "Netronome SmartNIC," https://www.netronome.com/blog/p4-programmability-for-the-netronome-agilio-smartnic/, accessed: 2018-02-20.

[13] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon, "P4FPGA: a rapid prototyping framework for P4," in *Proceedings of ACM SOSR*. ACM, 2017, pp. 122–135.

[14] P. Benácek, V. Pu, and H. Kubátová, "P4-to-VHDL: Automatic generation of 100 gbps packet parsers," in *Proceedings of the IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2016, pp. 148–155.

[15] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown, "Programmable packet scheduling at line rate," in *Proceedings of ACM SIGCOMM*. ACM, 2016, pp. 44–57.

[16] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches." in *Proceedings of NSDI*, 2015, pp. 103–115.

[17] H. T. Dang, H. Wang, T. Jepsen, G. Brebner, C. Kim, J. Rexford, R. Soulé, and H. Weatherspoon, "Whippersnapper: A P4 language benchmark suite," in *Proceedings of ACM SOSR*. ACM, 2017, pp. 95–101.

[18] W. Shen, M. Yoshida, K. Minato, and W. Imajuku, "vconductor: An enabler for achieving virtual network integration as a service," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 116–124, 2015.

[19] A. Lombardo, A. Manzalini, G. Schembra, G. Faraci, C. Rametta, and V. Riccobene, "An open framework to enable NetFATE (Network Functions at the edge)," in *Proceedings of IEEE NetSoft*. IEEE, 2015, pp. 1–6.

[20] S. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, T. Wood, M. Arumaithurai, and X. Fu, "NFVnice: Dynamic backpressure and scheduling for NFV service chains," in *Proceedings of ACM SIGCOMM*. ACM, 2017, pp. 71–84.

[21] F. Callegati, W. Cerroni, C. Contoli, and G. Santandrea, "Dynamic chaining of virtual network functions in cloud-based edge networks," in *Proceedings of IEEE NetSoft*. IEEE, 2015, pp. 1–5.

[22] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes." in *Proceedings of NSDI*, vol. 13, 2013, pp. 227–240.

[23] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "OpenNF: Enabling innovation in network function control," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 163–174.

[24] P. Zave, R. A. Ferreira, X. K. Zou, M. Morimoto, and J. Rexford, "Dynamic service chaining with dysco," in *Proceedings of ACM SIGCOMM*. ACM, 2017, pp. 57–70.

[25] D. Hancock and J. Van Der Merwe, "Hyper4: Using P4 to virtualize the programmable data plane," in *Proceedings of ACM CoNEXT*. ACM, 2016, pp. 35–49.

[26] C. Zhang, J. Bi, Y. Zhou, A. B. Dogar, and J. Wu, "HyperV: A high performance hypervisor for virtualization of the programmable data plane," in *Proceedings of International Conference on Computer Communication and Networks*. IEEE, 2017, pp. 1–9.

[27] ETSI, "Network Functions Virtualisation (NFV); Architectural Framework v1.1.1 ETSI GS NFV 002," 2013, http://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/002/01.01.01_60/gs_NFV-002v010101p.pdf [Accessed: 01.04.2018].

[28] W. Wang, Y. Liu, Y. Li, H. Song, Y. Wang, and J. Yuan, "Consistent state updates for virtualized network function migration," *IEEE Transactions on Services Computing*, 2017.

[29] S. Luo, H. Yu, and L. Vanbever, "Swing state: Consistent updates for stateful and programmable data planes," in *Proceedings of ACM SOSR*. ACM, 2017, pp. 115–121.

[30] N. Katta, M. Hira, A. Ghag, C. Kim, I. Keslassy, and J. Rexford, "CLOVE: How I learned to stop worrying about the core and love the edge," in *Proceedings of ACM HotNets*. ACM, 2016, pp. 155–161.

[31] M. Gao, B. Addis, M. Bouet, and S. Secci, "Optimal orchestration of virtual network functions," *Computer Networks*, 2018.

[32] S. Avallone, S. Guadagno, D. Emma, A. Pescape, and G. Ventre, "D-itg distributed internet traffic generator," in *Proceedings of International Conference on the Quantitative Evaluation of Systemss*. IEEE, 2004, pp. 316–317.

[33] "Scapy Library," https://github.com/secdev/scapy, accessed: 2018-06-10.

[34] "OpenStack Instance Live Migration," https://docs.openstack.org/nova/pike/admin/configuring-migrations.html, accessed: 2018-06-10.