TECHNISCHE UNIVERSITÄT DARMSTADT

SEAMLESS FLEXIBILITY IN HIGH-PERFORMANCE NETWORK FUNCTIONS VIRTUALIZATION

Am Fachbereich Elektrotechnik und Informationstechnik
der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines
DOKTOR-INGENIEURS (DR.-ING.)
genehmigte Dissertation

von
LEONHARD NOBACH, M.SC.,
geboren am 11. Mai 1986 in Kassel.

Vorsitz:	Prof. Dr. techn. Heinz Koeppl
Referent:	Prof. Dr.-Ing. Ralf Steinmetz
Koreferent:	Prof. Dr.-Ing. Wolfgang Kellerer
Tag der Einreichung:	3. Juli 2018
Tag der mündlichen Prüfung:	26. Oktober 2018

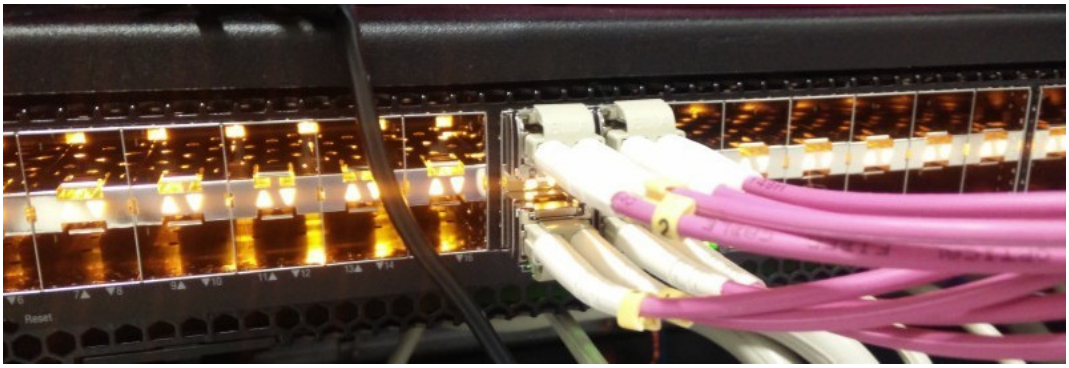
Hochschulkennziffer D17
Darmstadt 2018

Leonhard Nobach: *Seamless Flexibility in High-Performance Network Functions Virtualization,* Enabling Wide-Area Flexibility and Improving Resource Efficiency in NFV Infrastructures, Darmstadt, Technische Universität Darmstadt,

Jahr der Veröffentlichung der Dissertation auf TUprints: 2018
URN: urn:nbn:de:tuda-tuprints-81640

SEAMLESS FLEXIBILITY IN HIGH-PERFORMANCE NETWORK FUNCTIONS VIRTUALIZATION

LEONHARD NOBACH



Enabling Wide-Area Flexibility and Improving Resource Efficiency in NFV Infrastructures

November 2018

ABSTRACT

Communication network carriers are challenged to continuously deliver higher-performance, more adaptable network services to even lower costs. *Network Functions Virtualization (NFV)* is an architectural concept aiming to decrease costs and increase flexibility of a network infrastructure. In an NFV architecture, *network functions*, which are traditionally executed on specialized appliance hardware, are executed on standard, inexpensive, and general-purpose servers. Furthermore, NFV applies *cloud computing* principles to the network functions implemented for standard hardware, enabling elasticity, flexibility and a fast time-to-market.

For a sufficient flexibility, it is often desired that network function instances can be quickly moved between physical locations while they are in operation, which requires *seamless state migration*. Existing state migration mechanisms have been primarily designed for and tested in intra-datacenter situations. However, new concepts like carrier *edge clouds* and *fog computing* might require a state migration method for network function instances over long-distance links. The latters likely do not provide the throughput and latency available in a datacenter. We have identified that current methods can only migrate seamless in long-distance situations, if either the network function or the long-distance link is subject to low utilization. Furthermore, there are currently elasticity limits when using hardware acceleration for NFV environments. Due to the fixed set of commodity CPU and hardware acceleration resources on a computing node, either of the aforementioned resource types might become underutilized. Furthermore, the extraordinarily high performance of widely-available, inexpensive chipsets found in network switches could highly increase resource efficiency of network functions. However, the use cases of these chipsets are commonly limited in functionality, and it is unclear if a carrier-grade network function can be implemented by using them.

In this thesis, we propose a seamless migration mechanism for virtualized network functions, which reduces the state migration traffic compared to the state of the art by omitting redundant information. Our evaluation shows that if compared to the state of the art, the reduction of the migration traffic allows an almost *three-fold increase* of the network function instance's or the link's utilization during migration, while completing the migration in only one third of the time. We propose an architecture which meets elasticity demands of network function implementations requiring heterogeneous processing resources like FPGAs, commodity CPUs, or in-network processing. We furthermore propose a method to quantify the benefits of elastic FPGA provisioning. Finally we investigate the functionality of a widely-used switching chipset in the context of carrier network functions, and conclude that all essential features of a *Broadband Remote Access Server (BRAS)* can be implemented using it. Overall, we show that we can improve flexibility through enabling NFV state migration over long-distance links, as well as resource efficiency via increased hardware acceleration utilization.

ZUSAMMENFASSUNG

Betreiber von Telekommunikationsnetzen (Carrier) stehen vor der Herausforderung, immer leistungs- und anpassungsfähigere Netze zu geringen Kosten anzubieten. "*Network Functions Virtualization (NFV)*" ist ein Architekturkonzept, welches darauf abzielt, die Kosten zu senken, und die Flexibilität eines Netzwerkes zu erhöhen. In einer NFV-Architektur werden *Netzfunktionen*, welche traditionell auf spezialisierten Hardwaregeräten implementiert sind, auf handelsüblichen und günstigen – weil vielseitig anwendbaren – Servern ausgeführt. Desweiteren wendet NFV die Prinzipien des Cloud-Computing auf die auf Standardhardware implementierten Netzfunktionen an, und ermöglich damit Elastizität, Flexibilität und eine schnelle Markteinführung.

Für eine ausreichende Flexibilität is es oftmals wünschenswert, dass Netzfunktionsinstanzen schnell zwischen realen Ausführungsorten bewegt werden können während sie sich im Einsatz befinden, was *nahtlose Zustandsmigration* erfordert. Existierende Mechanismen für Zustandsmigration wurden mit einem Schwerpunkt auf Situationen innerhalb eines Rechenzentrums entwickelt und getestet. Jedoch können neue Konzepte wie *Edge-Clouds* (Cloudinstanzen am Rande eines Netzwerks) oder *Fog-Computing* ein solches Verfahren für eine Zustandsmigration über Weitverkehrsverbindungen erfordern. Letztere stellen aber wahrscheinlich nicht den Durchsatz und die Latenz zur Verfügung, die in der Regel innerhalb eines Rechenzentrums verfügbar ist. Wir haben festgestellt, dass aktuelle Verfahren in solchen Situationen nur dann nahtlos migrieren können, wenn die Nutzlast der Netzfunktion oder die Verbindungsauslastung gering genug ist. Weiterhin existieren momentan Elastizitätsgrenzen bei der Verwendung von Hardwarebeschleunigung in NFV-Umgebungen. Aufgrund der festen Menge an Standardprozessor- und Hardwarebeschleunigungsressourcen auf einem Rechenknoten kann einer der letzteren Ressourcentypen unterbelegt sein. Außerdem könnte die außerordentlich große Leistungsfähigkeit der breit verfügbaren und günstigen Chipsätze, welche sich in Netzwerkswitches befinden, die Ressourceneffizienz von Netzfunktionen erhöhen. Jedoch sind die Anwendungsfälle dieser Chipsätze hinsichtlich ihrer Funktionalität begrenzt, und es ist aktuell unbekannt, ob eine Netzfunktion mit ihnen implementiert werden kann, die den Bedürfnissen eines Carriers entspricht.

In dieser Dissertation schlagen wir einen Mechanismus für nahtlose Zustandsmigration virtueller Netzfunktionen vor, welcher den Datenverkehr für die Zustandsmigration im Vergleich zum aktuellen Stand der Technik reduziert, indem er redundante Informationen vermeidet. Unsere Auswertung zeigt, dass, verglichen mit dem Stand der Technik, diese Reduktion eine fast *dreifache Vergrößerung* der Auslastung der Netzfunktionsinstanz oder der Verbindung erlaubt, während die Migration in einem Drittel der Zeit möglich ist. Wir schlagen eine Architektur vor, welche Elastizitätsansprüche von Netzfunktionsimplementierungen erfüllt, die heterogene Ausführungsressourcen wie FPGAs, Standardprozessoren oder In-

Netzwerk-Verarbeitung erfordern. Daneben schlagen wir eine Methode vor, mit der der Nutzen elastischer FPGA-Bereitstellung quantifizierbar ist. Schließlich untersuchen wir die Funktionalität eines weitverbreiteten Switch-Chipsatzes vor dem Hintergrund von Carrier-Netzfunktionen, und schlussfolgern, dass alle essenziellen Eigenschaften eines *Breitband-Remotezugangsservers (BRAS)* mit diesem implementiert werden können. Im Allgemeinen zeigen wir, dass wir die Flexibilität durch die Ermöglichung von NFV-Zustandsmigration über Weitverkehrsverbindungen verbessern können, sowie die Ressourceneffizienz durch erhöhte Ausnutzung von Hardwarebeschleunigung.

CONTENTS

1 INTRODUCTION

The rise of cloud computing has significantly influenced the way in which we operate Internet services. Virtualization started to allow sharing physical resources of a server to operate virtual machines (VMs) on them, and *cloud computing* infrastructures now care about the elastic operation of virtual machines or other resources on thousands of servers, transparently distributed on datacenters (DCs) around the globe. Cloud infrastructures are expected to carry 94% of total DC workloads in 2021 [25]. The term network function (NF) originates from telephony [75] but can nowadays be generalized as a process which is modifying, inspecting, or making decisions based on data traffic in communication networks. Examples are firewalls, intrusion detection systems, or Internet Protocol (IP) telephony gateways. The concept of network functions virtualization (NFV) has emerged in 2012 at the *SDN and OpenFlow World Congress* in Darmstadt [114]. First, NFV is about running NFs on commodity server platforms, which are widely in use at the DC edge and are usually sold in high volumes to a relatively low price compared to network appliances. Secondly, NFV applies resource virtualization and cloud computing principles on NFs, for example by running x86-based processes in virtual machines or containers. With these virtualized network functions (vNFs), NFV targets to improve resource utilization, deployment time, elasticity, and flexibility.

An elastic and flexible NFV platform can adapt the resource supply or relocate processes to other resources during operation, and is especially desired in a network in which the demand for services changes over time, or in which the network infrastructure suffers from failures or benefits from upgrades. Such a platform might require changing the topology of the vNF instances executed on computing nodes. Therefore, vNF instances must move their state from one to another resource, but the vNF operation should continue without downtime in service delivery. However, the current state of the art for the aforementioned *seamless state migration* has limits when migrating high-performance NFs over weak inter-DC links. Furthermore, hardware acceleration (HWA) can solve performance challenges in NFV infrastructures, for example traffic in DCs will grow 3-fold from 2016 to 2021 [25], while latency must be reduced for novel applications. However, it is unclear how the provisioning of HWA should be organized to fully exploit elasticity, and how large the benefits of elastic provisioning are. There is also an uninvestigated potential to increase performance by exploiting hardware capabilities in network switches.

In this work, we contribute a mechanism that enables seamless state transfer of highly-utilized NFs even over small-bandwidth, high-latency network links. We also propose a reference architecture for elastic provisioning of HWA in NFV infrastructures, a method to identify the benefits of elastic HWA provisioning, and evaluate the capabilities of merchant switch silicon in bare-metal switches to accelerate a carrier-grade NF with highest demands on performance.

1.1 MOTIVATION

For a high-performance and low-cost NFV infrastructure, it should be carefully planned how the NF instance is connected to the network in which it shall operate. NFV placement [68] strategies make the decision about which computing node shall run the NF instance, usually with a special focus on optimizing network connectivity, guaranteeing certain network performance, minimizing link utilization, or a combination thereof. Automated NFV placement benefits from a software-defined networking (SDN) controller, which can automatically interconnect a placed vNF instance with endpoints, while providing a fine-granular traffic control on the forwarding devices. In 2021, 67% of DCs are expected to have partially or fully adopted SDN [25].

Compared to intra-DC traffic only growing by 23%, it is forecasted that DC-to-DC traffic will grow by 33% from 2016 to 2021 [25]. As DC-to-DC links suffer from significant latency by physical limits, and high-throughput links are expensive, a naïve placement would highly deteriorate NF performance and increase connectivity costs, which is also valid for inter-DC placement in cloud infrastructures [62]. Networks with user and device mobility, changing infrastructure conditions, and changing resource demand should use a *dynamic NFV placement*. This is especially relevant for new trends like carrier *edge clouds* and *fog computing* [10, 69], which place virtual NFs very close to the user, like at customer premises, mobile base stations, or on end-user devices, primarily in order to reduce latency, which currently averages on 46*ms* for mobile traffic in Western Europe [25]. After placing an NF instance on a node, a dynamic strategy can later decide that it should be moved to another node, or that a *split or merge* [140] operation shall be conducted to enable redistribution on multiple instances. Moving the NF instance requires its state to be transferred over a network, which leads to an additional bandwidth occupation of network links during migration. Some mechanisms move sessions or flows one-by-one between instances, we refer to this group of mechanisms as *flow migration* mechanisms. They require a frequent interaction with the SDN controller, which must be aware of every flow handled by an instance, furthermore, global state cannot be reliably maintained during migration.

Other mechanisms avoid the frequent per-session SDN controller interaction and only need one or two operations for traffic redirection, which we refer to as *instance migration*. The naïve approach is to *halt* the execution of the VM process, send a serialized snapshot of the current state to a destination, and resume the VM execution at the destination afterwards. However, the NF instance's downtime during transfer usually leads to an unacceptable service interruption. *Delta-based* VM migration mechanisms [26] significantly reduce downtime by continuing execution at the source, sending consecutive rounds of state which has changed during the previous transfer round (*state resynchronization*), while halting the VM only during the last round. Other resynchronization mechanisms use *deterministic replay* [57, 97], they capture external events (like packets) during snapshot transfer while not halting the source VM, send them to the destination instance, and update the outdated snapshot by replaying the events at the destination. Despite the advances in NFV instance migration, a recent survey confirms that the performance of instance migration over wide area network (WAN) links, like between DCs or edge clouds, is only insufficiently addressed by research [172]. We have found that existing mechanisms are not suitable for seamlessly migrating a low-latency and high-performance vNF instance over these links, because the state resynchronization traffic cannot keep up, which we have identified as being especially high in NFV.

Problem statement

To address NFV performance challenges, *field-programmable gate arrays (FPGAs)* have been proposed for NFV environments [13, 20, 55, 83]. However, first, FPGAs and other hardware accelerators can only solve *simple* operations efficiently, but many NFs also require executing a set of complex operations (like authentication phases) with low performance demands, which are more efficiently handled by the commodity central processing unit (CPU). Secondly, in state-of-the-art infrastructures, the elasticity of HWA provisioning is limited to the resources on the current computing node, which are bound to the local CPU. Thirdly, no method has been proposed to determine the *benefit* of an elastic provisioning of resources like FPGAs. Furthermore, in the networking area, *merchant switch silicon* application-specific integrated circuits (ASICs) are widely used in switches and other networking devices. Given the wide availability of the chipsets, they can be considered as commodity hardware like CPUs, but deliver a much higher performance than the latters. However, their flexibility to implement carrier-grade use cases is yet unknown, as they have been originally designed for only a very restricted set of networking tasks.

1.2 RESEARCH GOALS

The overall goal of this work is to improve the performance, resource utilization and flexibility of NFV infrastructures potentially distributed all over the world. Based on the current state of the art of NFV, we have identified two major goals which we want to achieve. We furthermore derive research questions (RQs) which are addressed by the contributions summarized in Section 1.3.

> **Goal 1:** Enabling NF instance migration in infrastructures interconnected over large distances (WAN links).

We need a mechanism that can seamlessly migrate NF instances which operate on intermediate traffic of latency-sensitive network applications, like voice or video conferencing, accepting downtimes in the order of a few milliseconds at maximum. The mechanism should operate over WAN links (with WAN-typical latencies) and be able to migrate instances which are processing traffic consuming even more than half of the available network link capacity between the source and destination instance. With such a mechanism, we would improve flexibility by enabling seamless NF instance relocation, or elasticity via seamless NF splits and merges over WAN links.

- **RQ 1.1**: How to design and implement a seamless instance migration mechanism for high-performance NFs over links with limited performance, using state-of-the-art hard- and software?

- **RQ 1.2**: What is the performance and cost benefit of the designed and the implemented instance migration mechanism compared to the state of the art under WAN conditions?

> **Goal 2:** Achieving higher resource efficiency using HWA in NFV.

NFV processing resource efficiency shall be improved in two different ways. First, we intend to increase resource *utilization* in an architecture comprising *heterogeneous* computing resources, for example commodity CPUs and HWA. Secondly, the potential regarding the *functionality* of merchant switch silicon to implement NFs with highest performance demands is of interest. If the functionality is given, the extraordinarily high *performance* of the silicon could fulfill such an NF service with just *one instance*, which would require a scale out to many servers in case commodity CPUs are used. Given that such a bare-metal switch is nearly as inexpensive and consumes approximately as much rack space and energy as a single commodity server, the efficiency can be highly increased.

- **RQ 2.1**: How must a generic NFV architecture be designed to efficiently support on-demand provisioning of heterogeneous processing resources?
- **RQ 2.2**: What are the benefits of elastic provisioning of HWA in NFV?
- **RQ 2.3**: In how far can merchant switch silicon fulfill functional requirements of a carrier-grade network function on a bare-metal switch?

1.3 CONTRIBUTIONS AND APPROACH

We address the two aforementioned goals and the related research questions with the following contributions, and briefly sketch our approach.

> **Contribution 1.1:** Proposal of the statelet method to reduce state resynchronization traffic of state transfer mechanisms. Design and implementation of SliM, an NF migration mechanism using the statelet method (RQ 1.1).

Because of the overhead of tracking memory deltas via shadow pages in memory (VM live migration), we choose deterministic replay for state synchronization, like previous works [57]. For NFs, replayed external events are commonly *packets*, which are duplicated like in *OpenNF* [56, 57] and used for replay at the destination. However, as only a small amount of information is commonly required for internal state change in an NF (like header fields) [14], a lot of information transferred for state synchronization is *redundant* (like payload irrelevant for the NF). We propose to identify the required information – called *statelet* – with help from the NF through an interface, which we also propose and specify in this work. The introduction of an interface requires adaptation of the NF implementation, the NF must be aware of the state synchronization mechanism. However, several other contributions to improve elasticity also introduce new interfaces to the NF [57, 140] and these efforts could be compiled to a single, standardized application programming interface (API) in the future.

We contribute Seamless Instance Migration (SliM), an instance migration mechanism using the NF's statelet interface for state resynchronization. We select a choice of state-of-the-art hardware and software technologies based on dataplane performance and relevance in the NFV community, and contribute a reference implementation of SliM based on these technologies. Furthermore, we contribute a C-based statelet interface for NFs. Details on the choice for a specific implementation platform can be found in Section 4.5.

> **Contribution 1.2:** Analysis and testbed evaluation of the *SliM* implementation under WAN conditions (RQ 1.2).

If a mechanism to seamlessly migrate NF instances has been found, we should evaluate its actual performance limits, especially during migration time, given a restricted bandwidth capacity shared between dataplane and migration overhead. Furthermore, we should evaluate the interdependency between performance and link capacity, or the maximum dataplane link utilization under which an NF can be migrated seamlessly.

To quantify the benefits of the statelet approach and SliM, we derive a mathematical model for analysis of the statelet approach and its comparison to packet duplication mechanisms. We also evaluate the performance and workload capacity limits of the SliM mechanism using the reference implementation in a lab testbed, aiming to achieve more realistic conditions than with the model used in analysis.

> **Contribution 2.1:** A reference architecture for elastic provisioning of HWA in NFV environments and a method to evaluate benefits of elastic provisioning (RQs 2.1, 2.2).

We propose an NFV infrastructure architecture based on commodity CPUs extended with HWA provisioning support. Here, we *pool* HWA resources in a DC, instead of keeping them bound to a certain computing node.

To assess benefits of elastic provisioning, we first evaluate performance capabilities of a NetFPGA board and a 2-core virtual machine on an x86 commodity CPU. Secondly, we describe and specify a model which allows us to determine cost benefits. Thirdly, in order to determine a quantified benefit, we apply the model on the obtained performance data, measured usage traces obtained from previous work [28], and current retail prices of the hardware used.

> **Contribution 2.2:** The assessment of merchant switch silicon to support the acceleration of a carrier-grade network function (RQ 2.3).

Merchant switch silicon can deliver highest packet-processing performance and is relatively inexpensive. However, as it has been originally designed for routing and switching only, the packet processing flexibility is limited, thus making it only applicable to a restricted set of use cases. For our investigation, we choose a hardware platform which has a very high market share, and select a broadband remote access server (BRAS) as a use case, as it must achieve very high performance, must allow for large scalability, and needs to inspect packet headers beyond just forwarding. As the platform's performance is beyond the evaluation limits of our testbed, we focus on a *functional evaluation* of the BRAS implementation.

1.4 THESIS STRUCTURE

The outline of our thesis is as follows. Chapter 2 provides an overview of NFV and further research and technology areas on which our work is based. It also provides an insight into industry-driven development and standardization activities related to NFV. The following Chapter 3 introduces and explains research work which is directly related to our contributions in more detail.

We then propose the statelet method and introduce SliM (Contribution 1.1) in Chapter 4. Here, we also describe our evaluation of SliM regarding performance and costs, and present the evaluation results (Contribution 1.2). We describe the reference architecture for elastic provisioning of HWA (Contribution 2.1) and describe the model to evaluate its benefits and the obtained results in Chapter 5. We present our BRAS implementation on merchant switch silicon and evaluate the functionality which could be implemented in Chapter 6 (Contribution 2.2). In Chapter 7, we conclude the thesis with a resume of the contributions, the consequences of our results, and an overview of possible future work.

2 BACKGROUND

> Parts of the following chapter summarize the findings of a previously published survey of the author of this thesis on NFV [124]. Section 2.11.1 contains citations from the author's publication [119] which are not explicitly marked. For more information, refer to Appendix B. Cited figures are explicitly marked.

Before describing our contributions, this chapter provides an overview of research and technology areas which build the foundation of our work. Besides providing a starting point to get familiar with NFV and related technologies, this section provides definitions and clarifications, because especially in the young research field of NFV, some terms are not consistently used in literature.

Chapter overview

Virtualization and cloud computing (Section 2.1) can be seen as one of the foundations of NFV. Furthermore, we provide an understanding of the term *network function* (Section 2.2). We then explain NFV in detail (Section 2.3), introducing its application areas (Section 2.4), methods to manage and orchestrate NFV with a special focus on standardization efforts (Section 2.6), its relation to SDN (Section 2.5), and the need for elasticity and state migration (2.7). We introduce the various research and development efforts to improve NFV performance on commodity CPUs (Section 2.8). We conduct a brief sketch of the role of FPGAs in networks (2.9) and the purpose of the Programming Protocol-Independent Packet Processors (P4) language (2.10), and conclude with an overview of the bare-metal switching concept (2.11).

2.1 VIRTUALIZATION

History of virtualization

Virtualization systems have already been proposed in 1970, where the term was mentioned together with a time-sharing system for a CPU [110]. An early platform for virtualization has been introduced with the IBM VM/370 [146]. However, for commodity processors, virtualization was hard to achieve without major modification of the guest operating systems. Virtualization platforms reached a larger user base with *VMWare* [31] and *Xen* [4], focusing on commodity processor virtualization with only minor modifications to the guest. Along with 64-bit CPU architectures, virtualization support has been built into many of the following processor generations [160], allowing guest execution without operating system (OS) modifications.

What is virtualization?

In general, the term *virtualization* can be defined as a form of resource management, where a *physical* resource (or a pool of physical resources) is provided as one or many *virtual* resources to resource consumers via a virtualization layer. Key idea is that the interface to the virtual resource implemented by the virtualization layer resembles the interface to the physical resource. Thus, a resource consumer does not require different interfaces whether accessing the physical or the virtual resource. For example, a physical hard disk connected by a Small Computer Systems Interface (SCSI) can be provided to multiple storage processes by granting each of them access to a separate virtual SCSI, i.e. a virtual hard disk. The virtualization layer must accept reads and writes on the virtual SCSIs and implement a way to consistently map the reads and writes to separate regions of the physical SCSI interface.

From virtualization to cloud computing

A secure virtualization layer also enables *multi-tenancy*, so parts of physical resources can be granted to different consumers not trusting each other. Pay-per-use billing models for resources benefit from *automated provisioning* and releasing of virtual resources through the virtualization layer. If a tenant can frequently acquire and release the currently desired amount of resources in a pool in a very short time, we speak of *elastic* virtual resources. The provisioning of elastic resources through a virtualization layer, transparently to the physical properties of the resource, has paved the way for *cloud computing*.

By sharing a physical resource between multiple consumers through virtualization without the need of the consumers to be involved in the details of the resource sharing, resource utilization is envisioned to be improved, thus overall costs can be saved.

Virtual machines (VMs) and VM placement

A computing process usually requires processing, memory, storage and networking resources. If an OS process is executed on a virtual set of these resources, the latter can be called a *virtual machine*. An important aspect is the *location* of the physical resources used by the virtualization layer of a virtual machine. Usually, CPU and memory of a virtual machine are provisioned on the same computing node, as the high-bandwidth communication between these resources requires a very low communication delay. The lower performance demands on storage resources allow the virtualization layer to place physical storage in more distant locations and connect them over a network while encapsulating the storage interface (iSCSI, Fibre Channel).

2.2 FUNCTIONS IN COMMUNICATION NETWORKS

What is a network function (NF)?

A network function (NF) is a component of a communication network processing intermediate traffic in a specific way. The European Telecommunications Standards Institute (ETSI), playing a major role in NFV standardization, defines an NF as a *"functional block within a network infrastructure that has well-defined external interfaces and well-defined functional behaviour"* [44]. An NF can be implemented by a specific hardware and software system, like an appliance, software on a server, or a distributed system. The term NF, however, abstracts from its actual implementation in hard- and software, and should not be used as a synonym for the implementation.

Middleboxes

The term *middlebox* can be a synonym either for the NF or its implementation, as the term not consistently used in literature; it is often used for the NF [128]. However, the definition in Request For Comments (RFC) 3234 [22] associates the term to the "intermediate device" and proposes the term "middlebox function" for the NF. For the purpose of a precise taxonomy and to avoid confusion in literature, the term *middlebox* is therefore avoided in this work.

NFs and middleboxes – a demarcation

Network functions are manifold (a wide range of use cases is introduced in Section 2.4). They must be distinguished from network *forwarding* tasks, but sometimes it is difficult to draw a clear demarcation. For example, a router maintains routing protocols and takes forwarding decisions, and probably answers Internet Control Message Protocol (ICMP) and Dynamic Host Configuration Protocol (DHCP) requests. However, the actual task of forwarding individual packets, based on an already given routing table, could be considered as not being an NF but a forwarding task, while the IP packet is not processed[1]. The middlebox definition of RFC 3234 considers a process operating above the IP layer to be a middlebox, and a process below to be a *forwarding* task, but is still indecisive about the IP layer itself [22]. We can see that the definition problem can be solved by *decomposing* the aforementioned functions of a router into sub-NFs and forwarding tasks.

Network appliances – Advantages

NFs are traditionally implemented on *network appliances*, hardware devices dedicated to the NF which they implement. License contracts for network appliances often cover a bundle of hardware and software. If they do not, the separately licensed software is commonly bound to a certain type of device (or even to a specific device via its serial number). Furthermore, it is common that the hardware is not intended to run software from alternative vendors.

Bundling the NF implementation's hardware and software on an appliance can have the advantage of reducing complexity for the operator, as it does not require the operator to identify appropriate hardware, the vendor commonly ensures flawless interaction with the software. Furthermore, it is less complex for the vendor to guarantee certain performance and up-time properties for an appliance than for software, which might be executed on different hardware platforms with different impact on performance and availability.

Network appliances – Disadvantages

However, using appliances, the network is *inflexible to* required or desired *changes* in the network's service functionality or its quantitative service demand. If a network operator intends to provide a new service, existing appliances might not be able to provide the required configuration, either by restrictions in the software, or by restrictions of processing capabilities in hardware. Therefore, new use cases often require the installation of new hardware at a specified location, together with appropriate re-wiring. The procurement and on-site maintenance not only incurs high costs, it also requires careful planning in advance and increases the delay of enrolling the new service. This also applies to increasing or decreasing service capacities (elasticity) in order to meet changing demand for existing services at a specific location.

Given the required hardware design and production process, appliance-based networking also leads to a longer *time to market* for vendors, thus to innovation delay. Additionally, the aforementioned process is expensive and likely not affordable for many start-ups, shrinking the vendor market. Ultimately, appliances require specially-skilled personnel to *maintain* them, for example in case of hardware defects.

1 No address and port translation assumed.

2.3 NETWORK FUNCTIONS VIRTUALIZATION

Origin of NFV

Although the term NFV [44] has been conceived in 2012 by a consortium of carriers [114], the idea behind it has a longer tradition. Especially the concept of running network functions on commodity hardware has already been usual before, for example with open-source routers [64, 86] or firewalls [141], operating on bare-metal server hardware. However, with the increased interest of carriers, the NFV concept received benefit not only by standardization activities [40, 79], but also by a growing research community [111].

> **Definition 0.1:** A **vNF** is a network function implemented on virtualized resources.

> **Definition 0.2:** An **NF instance** is an NF implementation operating on resources in a specific location (e.g. a computing node). One or multiple NF instances can implement an NF, but have a distinct state.

NFV: NFs on standard servers

As already sketched in the introduction, NFV rests on two pillars: First, NFV operates NFs on *standard hardware resources* (servers), which are sold or leased in high quantities on the market for a large variety of computing purposes, not only for networking. The wide availability of hardware providers enables competition, thus the resources are relatively inexpensive and fast to obtain. Furthermore, hardware maintenance procedures do not differ whether operating NFs or other services on a standard-server platform, no specially-skilled personnel is required, and no special spare parts must be kept in stock. NF operation might even be outsourced to datacenter operators not specialized in network services. Finally, there are advantages on the software side: Many operating systems, software development kits, libraries and build tools are widely and freely available for standard servers. The development or customization of software is therefore assumed to be more convenient and can probably be achieved in less time.

NFV: Virtualization for NFs

Secondly, NFV operates NFs on *virtualized resources* and applies *cloud computing* principles to it (see Section 2.1). Thus, the NF can profit from *elasticity*, *faster provisioning*, and *location transparency*. High-performance NFs often have strict requirements on a virtualization platform's network input/output (I/O) and forwarding, opening up a wide research area (Section 2.8). Furthermore, the orchestration and management of vNFs is network-centric (for example their arrangement in *service chains* [9]), and can highly benefit from the presence of SDN in the infrastructure (Section 2.5).

NFV standardization organization

The standardization activities started in 2012 through an ETSI Industry Specification Group (ISG) [40]. The group is currently formed by over 300 members, and focuses on *interoperability* of the components in an NFV ecosystem. The results of the standardization activities can be found in various documents available online. In the "Release 3", various features are included [41], like the NFV Information Model (IM-NFV), accounting features (CHRG), automated deployment, orchestration, NFV management and connectivity (MANO, VEMOSS, NFVWAN, NFVO_ARCH), VNF network acceleration and hardware-independent acceleration (FASTSWITCH, ACCEL), security, updates and upgrades (NFV_SEC, SEC4SNC, SWUP, POLICY), VNF snapshotting (VNF_PHOTO), testing and benchmarking (CONF&IOP), hardware environments (HWENV), and many more.

NFV standardization results

The Internet Engineering Task Force (IETF) standardization activities are organized in several working groups publishing draft documents, for example the VNFPOOL (Virtual Network Function Pools) group [166] or a group focusing on service function chaining [73]. An NFV research group (NFVRG) has been also formed at the Internet Research Task Force (IRTF) [79], publishing draft documents, like for analytics and verification [89, 148], to unify carrier and cloud networks [109, 155], elasticity [136], policy-based resource management [51, 88], and service chaining [93].

2.4 APPLICATIONS FOR NFV

A variety of NFs are present in today's networks which can be virtualized, or already have been virtualized [43]. In this section, we give an overview over NFV use cases, which have been addressed both from research and industry.

Carrier clouds

NFV cloud infrastructures for the special tasks of a telecommunications carrier are referred to as *carrier clouds* [156]. As carriers must lower the costs for infrastructure operation, they can thereby benefit from increased *scalability* and *elasticity* of their network services. Instead of running at a third-party site, for example in the infrastructure of a public cloud provider, carrier clouds are operated by the carriers themselves. Thus, carriers can ensure security and privacy, a low delay due to the infrastructure operation in proximity to the subscriber, and can achieve a deep integration of their access technologies into the cloud infrastructure.

CORD

The Central Office Re-Architectured as a Datacenter (CORD) [27] is an open-source initiative to provide an NFV- and SDN-based reference platform to provide the entire spectrum of a carrier's access services within a datacenter built with commodity hardware [129]. To implement NFs, the CORD datacenter comprises commodity servers with high-performance network connectivity ($2 \cdot 40 Gbit/s$ Ethernet). Network forwarding is achieved by using bare metal switches, controlled via the OpenFlow Data Plane Abstraction (OF-DPA) interface [17], which we also investigate for implementing a carrier-grade NF in Section 6. A major difference of CORD compared to other cloud datacenters are the *I/O blades*. They are used to ensure compatibility of the commodity hardware to the *physical* access technologies which are in use, like Gigabit Passive Optical Network (GPON) or Data Over Cable (DOCSIS).

Fixed access networks

A use case in fixed access networks is the virtualization of *residential gateways (RGs)*. These devices connect the customer premises to the carrier's network and may offer a variety of services: (1) *Layer 3 forwarding* between the subscriber's and the carrier's network, (2) network address translation (NAT) and port address translation, (3) network management (Address Resolution Protocol (ARP), DHCP, Simple Network Management Protocol (SNMP)), or (4) application layer gateways for telephony. An approach is to move these services from the customer premises to the carrier's network, while only a forwarding device (like a Layer 2 or SDN switch) is located at the customer premises [33, 167]. The provisioning of these services is also known as *Edge-as-a-Service* [30].

There are also efforts to virtualize the BRAS [8], also known as broadband network gateway (BNG). The function terminates the subscriber links at the ISP's side and commonly provides all access services at Layer 3 and above. The NF can be disaggregated into tunneling, accounting, billing, reverse-path filtering and address assignment functions. Regarding tunneling, it might provide a *Point-to-Point Protocol (PPP) over Ethernet (PPPoE) access concentrator* function, an important NF especially in European carrier networks, where PPPoE is a usual protocol on the access link.

Mobile networks

Virtualization of the functions of *mobile access networks* is a trending topic in research and industry, especially in the light of upcoming fifth-generation wireless systems (5G) networks. In radio access networks (RAN), NFV can provide flexibility and IT convergence [117]. The baseband unit (BBU) is a component of a wireless base station, and has been suggested for virtualization on standard hardware [72, 102]. NFV also supports prototyping [116] in a wireless network, furthermore, it is possible to share its resources with NFV [115, 170]. Besides the access side, the network core (like the Evolved Packet Core in Long-Term Evolution (LTE)) has been subject to virtualization efforts [157]. In mobile networks, NFV placement optimization also concerns the question whether to fully virtualize NFs or to decompose them and implement them in an SDN controller managing a forwarding device [6]. Mobile networks are furthermore especially profiting from a coexistence of SDN and NFV, thus from a joint optimization of SDN configuration and NFV placement [5].

Network core

An important class of network functions in core networks are *translation functions*. Carrier-grade NAT can anticipate the scarcity of public IPv4 addresses by sharing a single public IPv4 address between multiple subscribers. With Dual-Stack Lite (DS-Lite), carriers can avoid operating any IPv4 network in the access and even the core. Here, residential gateways *translate* IPv4 packets in the subscriber network to IPv6 packets in the carrier network. The translation back to IPv4 can be conducted later at the point-of-presence (PoP) through an Address Family Transition Router (AFTR), before relaying the packets to other autonomous systems (ASes). The AFTR has been subject to virtualization efforts [80].

IP multimedia subsystems

It has been proposed to virtualize the *IP Multimedia Subsystem* [21], which is a system of various network functions for *voice* services The session border controller (SBC) [63] serves as an Application Layer Gateway (ALG) for voice services. It terminates Session Initiation Protocol (SIP) sessions, is expected to provide session traversal for a large number of RTP sessions together with security and transcoding features. Efforts to virtualize the SBC have been found as feasible, but the performance of dedicated hardware could not be reached [112].

Load balancers

To ensure scalability, popular services must be fulfilled by multiple network nodes. A use case for NFV are *load balancers*, which distribute requests for a service among these nodes and ensure that they are efficiently used. Simple load balancing can be implemented in the SDN and classified as a *forwarding task*, like balancing based on the least significant bits of source addresses. More advanced load balancers may associate clients based higher layer information, e.g. on HTTP header fields, and must be considered an NF.

Content Delivery Networks

Content delivery networks (CDNs) are a use case for NFV. *CDN caches* are usually set up close to actual or potential receivers of content, for example at an autonomous system (AS) close to the receiver's service provider, or even in the service provider's network. With NFV, CDN caches can be dynamically provisioned and released, which enables the CDN to flexibly react on the current content demand. CDN caches require a larger amount of storage resources, which makes them special in the NF ecosystem. A virtualized CDN cache can serve content at 10 Gbps (line rate) [91].

Security

Finally, NFV can serve security purposes. The *firewall* is a classic NF traditionally implemented on an appliance. When decomposing the firewall function, several basic tasks like stateless IP filtering can be solved by SDN-enabled forwarding [95]. Support for stateful Transmission Control Protocol (TCP)/User Datagram Protocol (UDP) processing is limited in an SDN[2], and deep packet inspection (DPI) usually cannot be achieved in the SDN and can be considered as a function. Another use case lies in the mitigation of distributed denial of service (DDoS) attacks. It has been found that NFV supersedes appliances regarding scalability, energy consumption, and fast deployment of new techniques to thwart DDoS attacks [90].

2.5 SOFTWARE-DEFINED NETWORKING AND NFV

NFV and forwarding tasks

As mentioned in the previous sections, NFV can be distinguished from forwarding tasks. Traditionally, forwarding in a network infrastructure is done by *switches*, appliances equipped with a CPU and a forwarding ASIC. The CPU operates a *control plane*, which instructs the ASIC how to forward packets. The ASIC then reacts on incoming packets with forwarding to a specific port (group), dropping, or forwarding to the CPU for further inspection, These operations are tasks of the *data plane*. The traditional CPU/ASIC combination ensures forwarding performance and configurability.

Principles of SDN

However, the control plane on a switch CPU is tightly coupled to the appliance, and is commonly vendor-specific. Changing a configuration of a networking device therefore requires adopting a vendor-specific interface, hardening the interoperability of networking devices. SDN is a paradigm to overcome this [87]. SDN is about (1) separating the control plane and the data plane by a standardized and commonly vendor-independent interface, and it is about (2) applying software engineering principles to the control plane design and implementation.

SDN controllers

If the planes are separated by a network protocol like *OpenFlow*, multiple spatially-separated data planes can connect to a single, logically-centralized controller on commodity-hardware, and the latter can coordinate forwarding in the entire infrastructure with a high flexibility. SDN controllers are a key enabler for *network virtualization* [24] [9], which can slice the resources of a physical network infrastruc-

2 In OpenFlow this requires reactive flow modifications per socket.

ture into multiple isolated and virtual views of the network, which can be provided to different tenants.

OpenFlow – Capabilities and Limits

Although the OpenFlow protocol is extensible (OXM fields), it is intended for match and action operations up to Layer 3 and some Layer 4 fields, which are desired for packet forwarding. The problem is aggravated by limited support of available OpenFlow operations in hardware, for example, many ASIC abstractions (e.g. OF-DPA) do not support IP address rewriting [17]. Therefore, OpenFlow and other SDN south-bound protocols are not suitable for implementing arbitrary network functions. However, OpenFlow can be used for accelerating certain forwarding tasks of an NF. For example, a NAT NF or a firewall might add flows for newly established sockets through the SDN controller, thus *offload* performance-critical forwarding tasks.

How can SDN help NFV?

NFV applies cloud computing principles to network functions to achieve highest flexibility and scalability. The latter properties require automated management of NF instances, like selecting appropriate resources for NF instances (NFV placement), starting and stopping instances, migrating them to other resources, or splitting an instance into multiple ones or merging multiple instances in a single one. Especially the harsh network requirements of NFV demand that forwarding keeps up with the flexibility of NFV. SDN introduces software APIs to *define* network forwarding in infrastructures. Therefore, the task of an SDN controller can be, for example, to provide a path between two endpoints, ensuring certain performance guarantees and isolation. The SDN controller then reserves the resources on the links (like bandwidth), and creates forwarding rules on the intermediate switches to establish the path. SDN can therefore not only automate the virtual “wiring” of NF instances, it can also quickly react on changes in NFV placement by committing the appropriate forwarding changes to the network.

2.6 NFV MANAGEMENT AND ARCHITECTURES

ETSI NFV architecture

Figure 1 provides an overview of the NFV reference architecture proposed by the ETSI. The architecture is split into the NFV Management and Orchestration (MANO) and the operational NFV infrastructure (NFVI). The two domains communicate using interfaces on different layers for *operations and business support* (Os-Ma-Nfvo), NF management (Ve-Vnfm-vnf), and infrastructure management (Nf-Vi). In MANO, we distinguish the vNF Manager, coordinating the behavior of a specific NF [47], from the virtualized infrastructure manager (VIM), coordinating particular NF instances, their resources, and the network in the infrastructure.

Simplified architecture

For the purpose of this thesis, we focus on the lowest layer (VIM and the NFVI), where we apply our contributions. To better describe the components here, we use a different terminology in which the VIM is split into an *NFV controller* and an *SDN controller* (Figure 2). In our architecture, a logically-centralized NFV controller is responsible for coordinating NF instances or conducting actions on the hypervisor. This includes starting/stopping VMs, splitting and merging them, or provisioning resources for them. It delegates all *forwarding tasks* to the SDN controller, which

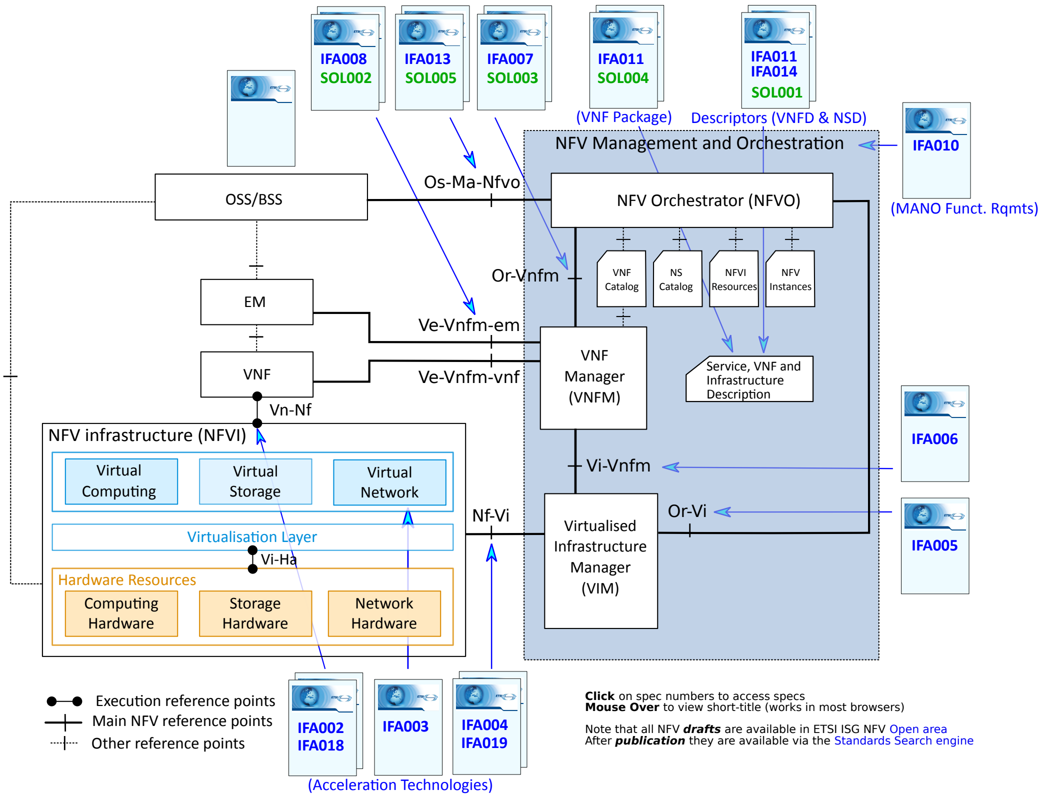


Figure 1: ETSI NFV Architecture Overview with relevant standardization documents. [39]

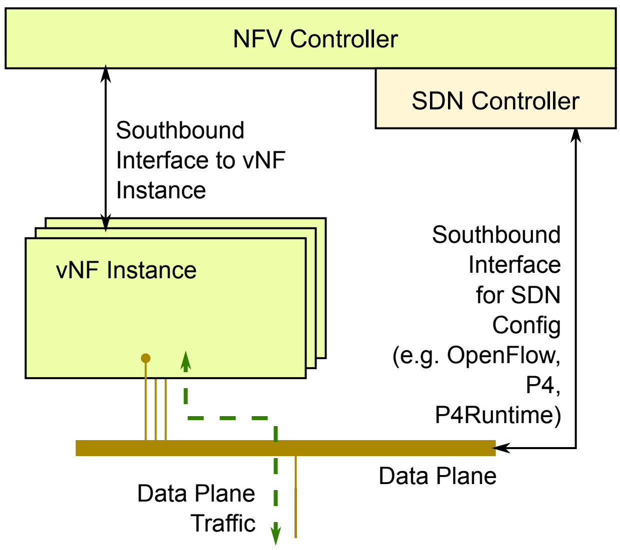


Figure 2: Architecture model used in this thesis. [122] (adapted)

ensures that packets inserted at a specific location in the network reach their intended destination.

A logically-centralized controller does not require to be executed on a single, physically centralized node. To avoid single points of failure, controllers can be operated redundantly on multiple devices (for example in hot- standby mode). Furthermore, peer-to-peer networks can provide fail-safe services without a centralized instance using the peer-to-peer paradigm [151], this can also be used to provide NFV instance control [52].

2.7 NFV ELASTICITY

What is elasticity?

A desired property of NFV is to enable *elasticity* of the NF services. According to Herbst et al., elasticity is "commonly understood as the ability of a system to automatically provision and deprovision computing resources on demand as workloads change" [66, 67]. On a computing node, it is possible to grant more CPU time or increase the memory resources (memory ballooning) assigned to an NF instance. However, it is not possible to increase resources beyond this limit if the NF capacity demands supersede the capacity of a single computing node. Therefore, the NF workload must be distributed between multiple computing nodes, referred to as *scale-out*. Likewise, the number of computing nodes executing an NF can be reduced if service demands decrease, referred to as *scale-in*.

> **Definition 1.1:** A flow-based NF processes one or multiple flows. Here, a *flow* is a set of packets processed by an NF which are usually related to each other by their source, their destination, or other properties relevant to the NF. A *flow state* is a subset of an NF's state used while processing packets of a specific flow, and which is not used while processing any other flow. Usually, an NF also maintains *global state* used among all flows, and sometimes *shared* state used among several, but not all flows.

NF splits and merges

When operating an NF on multiple scaled-out instances, ingress traffic must be distributed among them (load balancing). This is a non-trivial task, as packets must always reach the instance maintaining the appropriate state. For flow-based NFs, this means that packets belonging to a certain flow must reach the NF instance maintaining the respective flow state (e.g. interior and exterior packets of a Layer 4 socket in a firewall). The situation even becomes more complex with a dynamic set of NF instances, when *splitting* an NF instance into multiple ones together with the required SDN forwarding rules, or *merging* the latters into a single instance [140]. Splits are used if an instance cannot claim sufficient resources on a single node and must be distributed to multiple instances. Merges are triggered when multiple instances are under-utilized enough so that they can be combined in a single instance.

Split and merge strategies

To split a flow-based NF instance, two approaches are available:

- We start a new NF instance without any state, and only redirect packets from new flows to the new one. With this approach, it is not required to transfer the

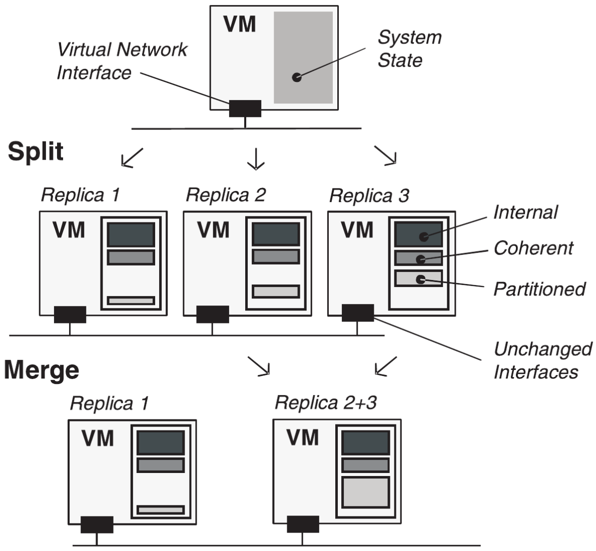


Figure 3: Splits and merges of NF instances. [140]

state of existing flows to other NF instances. However, this approach requires more time, as it must wait while new flows appear. Furthermore, it requires per-flow forwarding rules in the SDN, which could exceed the maximum flow table sizes of today's SDN switches.

- We split the flows processed by an existing NF instance so that we can use a scheme of *flow spaces*[3] to redirect packets to the instances using a few SDN rules only [57]. However, this requires an immediate transfer of the state of an entire flow space (a *snapshot*) to another instance.

The latter method requires *state migration*, which is also required for splitting NFs which are not flow-based (e.g. an intrusion detection system (IDS) detecting coordinated attacks), or NFs which have flows with a shared state which must be moved to the same target instance.

2.7.1 *NFV State Migration*

Being able to move state to other locations increases the flexibility and elasticity of an NFV infrastructure. First, it is possible to relocate entire NF instances using *full migration*, for example to bring them closer to a user, or to optimize the placement after changes in the network. Secondly, it is required by aforementioned splits and merges [57, 140] in the form of *partial migration*.

State migration

A full migration always transfers the entire state to a new location, and redirects the entire traffic to the new instance. When doing partial state migration however,

Full and partial migration

3 Examples for flow spaces: Wildcard matches on the least significant bits of a source IP in the one direction, and on the destination IP in the other direction. This way, packets are distributed to NFs based on their last bits, and it is ensured that returning packets always reach the same instance which maintains the appropriate flow state.

only a part of the NF's state is transferred to another node to take up operation in a new instance, while the rest remains in operation on the current one. Furthermore, only the part of the traffic is redirected to the new NF instance which is associated to the partial state which has been transferred. Mechanisms for splits and merges require a deep integration into the NF, as they must be aware about which state is used during packet processing, how the state can be split, and how the packet stream can be separated into flow spaces. An overview of VM and NFV state migration methods is provided in Section 3.1.

2.7.1.1 *Seamless State Migration*

Motivation

While many communication services like web services or downloads are tolerant against latency or packet loss, stricter requirements may apply to multimedia services and -protocols. For example, depending on the application, audio and video data must be processed in a certain period of time, and must be transferred before the aforementioned period is over. This also implies that the network must ensure it meets the required guarantees. [150].

Seamless Migration

NFV state migration bears the risk to cause *packet loss* and *jitter* in the network during the migration process. However, it is desired that negative effects of the state migration are not perceived by the user, especially in highly dynamic NF placements this would lead to a high deterioration of quality. Therefore, it is important that a state migration mechanism operates *seamless* when the NF instance operates on today's real-time traffic in the network, like Voice-over-IP (VoIP) and IP Television (IPTV).

> **Definition 1.2:** A state migration process operates *seamless* regarding a specific service or an application, if the service consumer or the application user does not perceive service disruption caused by the migration process.

What is considered as seamless?

For example, the International Telecommunication Union (ITU) recommends a jitter of less than $50ms$ during the play-out of an IPTV stream, and a "maximum duration of a single-error event" (duration of an isolated packet loss burst) of $< 16ms$ [76], which should be the maximum accepted downtime. The ITU also proposes a jitter buffer of $60ms$ on a VoIP stream [78], and recommends that VoIP receivers can cope with a delay variation of up to $50ms$ [77]. We therefore consider a state migration incurring a jitter below $50ms$ and an out-of-service time below $16ms$ to be seamless. In our work, we nevertheless aim to minimize jitter and downtime, to be able to cover future use cases, as well.

2.8 NFV PERFORMANCE

Motivation

Besides flexibility and elasticity, vNFs should provide high *network performance*. During vNF implementation, the performance demands on a vNF must be carefully considered, otherwise, the complexity of today's CPU architectures can deteriorate performance, or make performance unpredictable. Beyond the classic metrics for NF performance, *delay*, *jitter*, and *throughput*, NFV introduces new performance metrics relevant in dynamic environments: the *startup* or the *migration time*.

Performance challenges in NFV

When using dedicated hardware for NF implementation, it is possible to adapt the hardware to the NF, and dedicate guaranteed resources to NF processing. However, NFV introduces various *overhead* caused by not only virtualization, but also by the operating system and the nature of the general-purpose processor. For example, when using multiple *10 Gigabit Ethernet* interfaces on a computing node, it is not possible to reach line rate when processing small packets [105]. Programming disciplines e.g. require that packet buffers are only copied as few times as possible in memory (to increase throughput), or that packets are not unnecessarily waiting in queues (to decrease delay and jitter). Another problem is jitter incurred by virtualization overhead [164].

Section overview These problems have been subject to a variety of research on virtualization performance [71, 106, 107, 145], which we address in these section. We distinguish between research on *virtual packet forwarding*, *hardware-assisted packet forwarding*, *network stack offloading*, *VM network I/O optimization* [54] and the *reduction of OS complexity*.

2.8.1 *Virtual Packet Forwarding*

The task of *virtual packet forwarding* refers to forwarding packets on the same hypervisor between VMs and physical interfaces, implemented in software. For example, packets tagged with a specific Virtual local area network (LAN) (VLAN) arriving on a physical interface shall be forwarded to a specific VM, or traffic from a VM shall be passed to another VM on the same node. The software responsible for virtual network forwarding is referred to as a *virtual switch* or *vSwitch* (Figure 4 a).

Linux support Linux has inherent support for network forwarding via its *bridge* support, which can be used for virtualization environments. Upon request, Linux can instantiate bridges, which are basic Data Link Layer (L2) broadcast domains between selected ports, either physical ports of the hypervisor or virtual ports of the VMs. For VLAN support, a bridge can be assigned to VLAN sub-interfaces, or ports in bridges can be VLAN-filtered[4]. The *MacVTap*[5] adapter (Figure 4 b) implementation in Linux is a combined forwarding and I/O driver which can dispatch traffic from physical interfaces to VMs, however, it does not support bridging. The straightforward Linux solutions are restricted to basic L2 forwarding and thus lack flexibility for SDN integration.

Open vSwitch Open vSwitch [130, 131] introduces support for flexible traffic management in Linux. On the one hand, it preserves classic bridging and VLAN support and works by assigning bridges between virtual and physical interfaces. On the other hand, these bridges can be rendered to an SDN controller via protocols like OpenFlow or Open vSwitch (OVS) Database (OVSDB). Furthermore, it provides link aggregation and quality of service (QoS) policing. On a single-core system, Open vSwitch has been demonstrated with 1.8 million packets per second (Mpps), in contrast to

4 Discovered in the Proxmox Virtual Environment.
5 http://virt.kernelnewbies.org/MacVTap

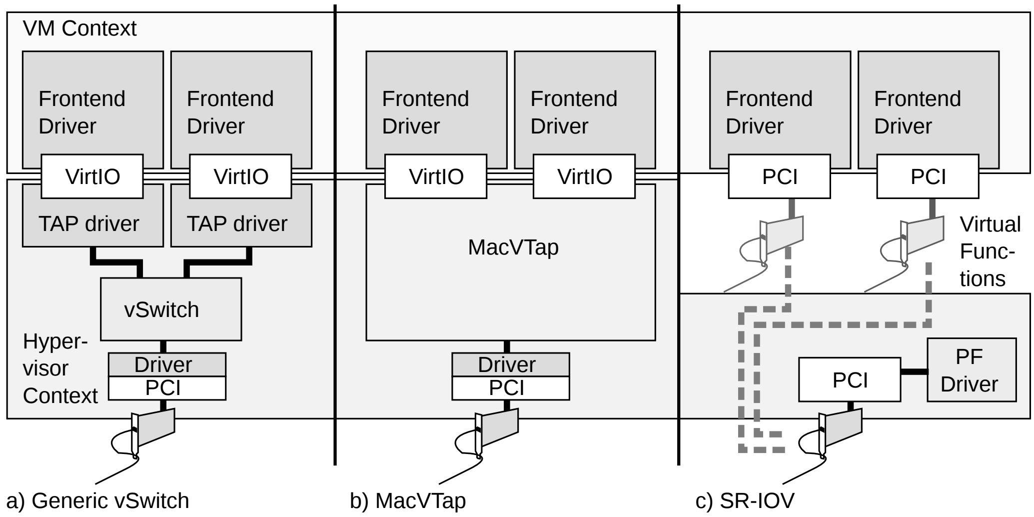


Figure 4: Different architectures for VM network forwarding and VM network I/O. [124]

the Linux bridge with only 1.1 Mpps [38]. However, the results show differences between the versions of Open vSwitch.

VALE and netmap

Performance can be further increased by bypassing the Linux kernel. The *VALE/netmap* software switch [143] is intended for high-performance packet forwarding *between* VMs, but it is restricted to basic L2 forwarding and does not support increased forwarding flexibility. The authors demonstrate a better performance than Open vSwitch in their evaluation, however, they use a modified Open vSwitch version to support netmap.

Zero-copy forwarding

A *packet buffer* is a memory region allocated to contain the bytes of a network packet. Copying large amounts of data in memory has a negative impact on CPU utilization [161]. Therefore, the number of packet buffer copies in memory should be minimized. If the packet buffer is not even copied once, we speak of *zero-copy* forwarding. Here, the network interface controller (NIC) writes the packet contents via Direct Memory Access (DMA) into the buffer in the final memory region. After the NF processes only relevant parts of the buffer (for example headers), the NIC reads the processed packet from the same memory region to transfer it.

Inter-VM Shared Memory

Achieving zero-copy forwarding in a *virtualized* environment incurs additional challenges, as packets must be passed from, to, or between VMs without a `memcpy` operation between host and guest memory. Therefore, zero-copy forwarding requires modifications in the vSwitch, the hypervisor, and the network device driver in the guest. The Inter-VM Shared Memory (IVSHMEM) can be enabled in Open vSwitch [135]. Inter-VM zero-copy forwarding techniques can however arise security concerns, as multiple, possibly untrusted entities may have full access to shared memory.

2.8.2 *HW-assisted Packet Forwarding*

Instead of using a software-based virtual switch for packet forwarding between VMs and hardware interfaces, we can offload packet forwarding to dedicated hardware. This relieves the CPU from this task and has the potential to speed up network performance [99].

PCI Passthrough

The Peripheral Component Interconnect (PCI) interface is a prominent standard for I/O on today's computers, and for sure the most important standard for operating network interfaces on a virtualization platform. The bus-based interconnect standard allows PCI devices to directly operate on the memory of the host, which is referred to as DMA. On platforms with an I/O Memory Management Unit (IOMMU)[6] and appropriate PCI bridge support, it is possible to securely grant a device on the host's PCI bus to a VM, even with full DMA support, which is referred to as PCI passthrough (PCI-PT).

Single-Root I/O Virtualization

The *PCI Special Interest Group (PCI-SIG)* has furthermore specified the Single-Root I/O Virtualization (SR-IOV) standard. A hardware PCI device, for example a NIC, can appear as multiple PCI devices to the hypervisor, so-called SR-IOV virtual functions (VFs). These VFs each have own I/O queues and an isolated PCI memory region, and can be arbitrarily granted to VMs using PCI-PT. Furthermore, a physical function (PF) is provided to the hypervisor to configure the hardware device, and to configure forwarding. Despite the vast performance gain of SR-IOV, its downside is the restricted flexibility compared to software-based solutions. When using SR-IOV, network forwarding is bound to the features provided in hardware, which are restricted to traditional, VLAN-based L2 bridging. In our tests, we even had problems to enable inter-VM traffic without an established link on the physical cable connected to the card. Furthermore, we experienced strict Medium Access Control (MAC) address dependency of a VF, thus we could not operate it in promiscuous mode.

Virtual Machine Device Queues

VM Device Queues (VMDq) are subject to a similar principle like SR-IOV, but are used for a different purpose: parallelization. Instead of creating multiple virtual interfaces for different virtual machines, it creates an interface with multiple packet queues where packets arrive and are picked up. As each queue can be exclusively used by a CPU core, multiple CPU cores can operate on these packet queues *in parallel*, thus it is not necessary to dedicate a separate thread to dispatch the load to multiple queues.

2.8.3 *Network Stack Offloading*

For smaller packets, current network stacks in today's operating systems cannot reach line rate [105]. Therefore, several efforts have been made to circumvent the network stack of the operating system.

Data Plane Development Kit

The Data Plane Development Kit (DPDK) [23] is a fully-fledged software development kit for the C programming language, which can be used to directly access

6 Known as "VT-d" on Intel platforms

network interfaces without involving the kernel. Only a restricted set of network cards are supported by DPDK. Upon startup, the DPDK application claims direct access to PCI devices via the `uio.ko` or `vfio-pci.ko` kernel modules, during packet I/O, the kernel is not involved anymore. The DPDK supports raw access to the packet bytes in the buffers, and does not implement a network stack like IP or TCP. However, it provides access to the network interface's queue management (VMDq) and hardware acceleration features like checksum calculation, if supported by the NIC [35]. A similar project is the Open Data Plane[7], which however focuses more on the API standardization.

In the opposite way, an NF implementation in the userspace may also offload parts of the NF functionality to the kernel [145]. Another strategy is to offload functionality traditionally implemented by the OS to dedicated hardware in the NIC [134].

2.8.4 *VM Network I/O Optimization*

Context switching in VMs

In virtualization environments, we distinguish between *full virtualization* and *paravirtualization*. Full virtualization supports running unmodified guest operating systems, no changes must be made to the guest to virtualize it. The compatibility must be paid by a very high performance deterioration, as emulating a "real" hardware device requires additional computation overhead for every I/O operation, e.g. by frequently changing between host and guest mode, referred to as *context switching* [7, 171]. Paravirtualization requires special device drivers, thus it is not transparent to the guest. Adapting the drivers to the virtualization environment however enables the opportunity to highly increase performance. *VirtIO*[8] is a common device driver for paravirtualized I/O in Linux.

Poll-mode drivers – avoiding interrupts

Another method to reduce context switching is the avoidance of *interrupts*. Normally, an interrupt is generated upon every packet entering the network interface. The interrupt causes a context switch, as the hypervisor must hand over the packet to the VM. The performance-consuming behavior can be avoided by using *poll-mode drivers*. Using these drivers, the interfaces do not announce new packets with an interrupt, but the drivers periodically check for new packets to be received. The periodic polls consume a large amount of CPU resources[9] during idle times, however, the periodic job pays off when the NF instance is sufficiently utilized.

Reducing impact of interrupts

Despite poll-mode drivers, several works on modifying the interrupt strategy exist. An alternative is given by *interrupt coalescing*, delaying interrupts until a threshold has been reached [34]. *Exit-less interrupts (ELI)* and the *exit-less virtual I/O system (ELVIS)* avoid VM exits [60] by using a shadow interrupt descriptor table (IDT) delivering the interrupts directly to the VM. The *New API (NAPI)* [96] uses a combination of interrupt and polling [54]. The NAPI does not have inherent multiprocessor support, but a multi-threaded NAPI has been proposed [70].

Performance comparison

7 http://www.opendataplane.org
8 http://www.linux-kvm.org/page/Virtio
9 If we reduce the poll frequency, latency will suffer.

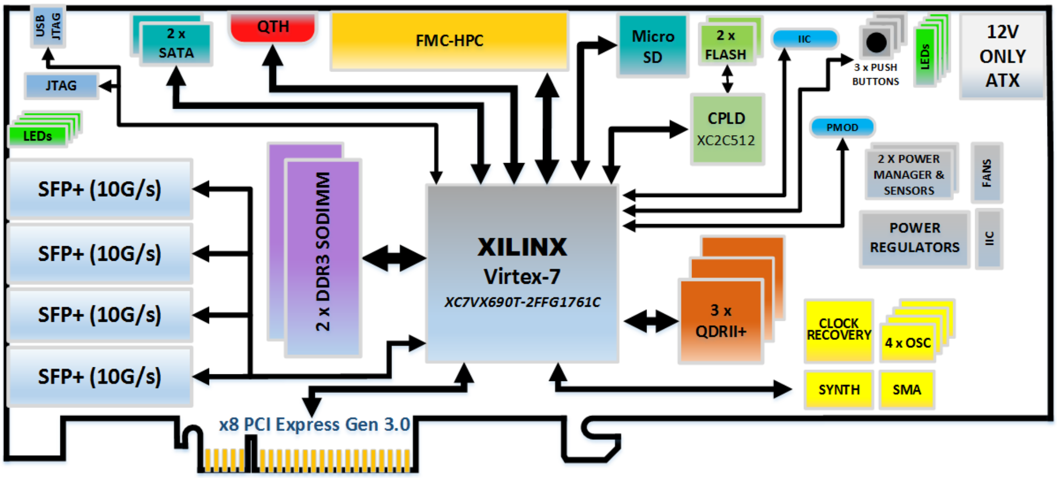


Figure 5: The NetFPGA SUME platform. [174]

Throughput comparisons [74] show a clear advantage of hardware-assisted technologies. These are followed by paravirtualization (vhost-net, VirtIO), while fully-virtualized interfaces (e1000) achieve lowest throughput.

Besides avoiding the operating system kernel for network I/O, a strategy is to reduce the complexity of the operating system and its kernel. One of these approaches is ClickOS [106, 107], which provides a light-weight unikernel including the Click modular router. It has been shown that it is feasible to quickly boot unikernels on demand if a network request arrives [103].

2.9 FPGAS IN NETWORKS

To achieve a high and predictable network performance, network appliances are often relying on ASICs. However, the production of a customized chip incurs very high costs, and their low flexibility is not compatible with the principles of NFV.

FPGAs are a flexible, high-performance alternative. FPGAs can be flexibly programmed, but instead of executing a procedural program on them, the functional behavior of a *gate logic* is mapped to the FPGA silicon. Therefore, they behave like ASICs, and their performance is assumed to be higher than the one of commodity CPUs in literature regarding many use cases [20, 83]. However, as they emulate a programmed gate logic behavior with a physical gate logic, their clock rate is usually lower than today's high-performance ASICs. Furthermore, the number of logic elements which can be used are limited. Software development for FPGAs is done using a hardware description language (HDL), which can be synthesized and converted into a *bitfile* that can be loaded onto the chip. Depending on the use case, both the performance and flexibility of FPGAs can be located somewhere between commodity CPUs and ASICs. Modern FPGAs support *Partial Reconfiguration*, which is a feature to reconfigure gate logic regions on-the-fly, while other gate logic regions are in operation, which has already been used to virtualize FPGA resources [20].

In this thesis, the NetFPGA platform [174] is used for prototyping of an FPGA-based NF instance (Section 5). The platform is equipped with a Xilinx Virtex-7 FPGA, which is directly connected to four 10 Gigabit Ethernet interfaces (SFP+ slots).

2.10 THE P4 LANGUAGE

Programming Protocol-Independent Packet Processors (P4) [11] is a language to *describe* a packet processing pipeline. P4 assumes a table model based on *matches* and *actions*, similar to OpenFlow. However, in P4, it is possible to define arbitrary headers – even in the application layer – and how they are parsed. Furthermore, the customization of table actions is possible, like header modifications or the selection of output ports.

Two major versions of P4 exist: $P4_{14}$[10] and $P4_{16}$, which have large differences in syntax and semantics. P4 is a language defining the pipeline *structure*, like the type of matched fields and the behavior of actions (like the command CREATE TABLE in SQL). A P4-defined pipeline is commonly fixed during operation, and the match-action table entries must be populated with entries during runtime, which is out of scope of P4.

2.11 BARE-METAL SWITCHES

Motivation

There is a large market of network switches, many brands provide devices with a broad set of different features. Commonly, such a switch is supplied with a vendor's proprietary software for its CPU, which allows an administrator to log in and configure the device using a command-line interface. The software on the CPU then renders all relevant changes to the forwarding hardware. Even switches capable of OpenFlow require a software on the switch which terminates the OpenFlow protocol and translates changes in the OpenFlow tables into a format required by the ASIC. Despite the variety of switch brands and types, the chipsets are very similar and are delivered by only a small number of vendors. Therefore, a number of switch vendors launched a *hardware-only* approach by selling switches with the *same* chipset types as in traditional switches, but without an operating system.

> **Definition 1.2:** A **bare-metal switch** is a switch supplied without an operating system.

Early adopters

With bare-metal switches, the operator can either choose from a set of available software products or customize the entire switch operating system based on the own needs. Bare metal switching received first larger attention in 2014, where Facebook launched its Open Switching System (FBOSS) [48] at the Open Compute Project (OCP) Engineering Summit. Likewise, Microsoft has proposed the Linux-based *Azure Cloud Switch (ACS)*. To get started as a developer, open-source communities like *Open Network Linux* have emerged, and a leading ASIC vendor commits drivers

10 Previously know as P4 1.0, the version P4 1.1 has been deprecated.

and APIs to the community [17]. Despite the advances in industry, the bare-metal switch *research* field is very young.

2.11.1 *Bare-Metal Switching and SDN: What is the Difference?*

Principle

Bare-metal switching is actually the SDN concept driven very far and to a low level. Most software-defined network controllers are commonly relying on standardized, south-bound protocols like OpenFlow to tell switches what to do. However, SDN switches then translate commands of these standardized, south-bound protocol commands into changes in the ASIC forwarding tables (Forwarding Information Base, FIB) or Ternary Content-Addressable Memory (TCAM).

Programmability

Bare-metal switches generally do not support any standardized south-bound protocol out of the box, if such a protocol is desired, it must be implemented in the operating system first. An attempt to achieve this on a bare-metal switch is the publicly available OF-DPA [17], (Section 6.1.1) in combination with the *Indigo OpenFlow agent*. Although OF-DPA is standardized, the applied table restrictions to OpenFlow are very narrow, and likely will reject an OpenFlow control plane not specifically adapted to OF-DPA. A proprietary alternative is the PicOS switch operating system [132], which aims at full OpenFlow or OVSDB compatibility.

Architecture and Performance

One idea behind the bare-metal-switching concept is that instead of standardizing the network communication protocol between the controller and the switch, the local forwarding API of the switch is standardized, and the SDN controller is extended to the switch CPU instead of being restricted to a remote server. This *distributed* concept has the potential to improve dataplane reaction times compared to delayed controller links (like when replying to a packet event or enabling forwarding upon a packet event), to reduce network traffic due to controller interaction, and to improve resilience of the control plane (if a controller or the link to it fails). Unfortunately, at the time of writing, there is no widely-supported vendor-neutral API to manage the ASIC, likely due to hardware differences.

3 RELATED WORK

While the previous chapter has given an overview over NFV concepts, technologies, trends and standardization activities in general, this chapter provides an overview over work directly related to our research challenges. In this regard, the section is split into Section 3.1, introducing work related to our first challenge about seamless state migration, Section 3.2, referring to work on elastic provisioning of hardware acceleration for NFV, and Section 3.3, discussing previous work on bare-metal switches.

3.1 NFV STATE MIGRATION

Several NFV state transfer mechanisms migrate on a *per-flow* level, they split the state into very small parts which can be migrated independently and quickly (Section 3.1.2). Other state transfer mechanisms are *snapshot*-based. They require a larger snapshot of the entire instance state to be transferred from a source to a target. While the naïve approach pauses the vNF instance during snapshot transfer, more advanced approaches continue execution at the source and *resynchronize* the state at the destination with delta-based (Section 3.1.1) or deterministic-replay-based methods (Section 3.1.3). Resynchronization is not only relevant for relocation, splits, and merges, it is also required by many *failure recovery* mechanisms [29, 138, 139, 147, 165].

3.1.1 *Snapshots with Memory-Delta-Based Resynchronization*

Before the scene for NFV shaped, a first attempt of migrating the memory of a VM, while avoiding downtime required for memory state transfer, has been proposed by Clark et al. [26] with *VM live migration*. The proposed mechanism works on the hypervisor level and is transparent to the VM. To track ongoing changes of the VM to its memory state, the mechanism uses a feature of the Xen hypervisor which inserts *shadow page tables*. In a first round, VM live migration transfers a consistent snapshot of the VM's memory to the destination, and enables shadow paging while continuing with VM execution. When the snapshot transfer has been acknowledged by the destination, it is likely outdated, as the ongoing execution of the VM at the source will have changed memory since the snapshot has been taken. Therefore, the changed (*dirtied*) memory pages are now transferred and acknowledged in a step which is repeated in multiple rounds, assuming the *memory delta* between two rounds becomes smaller with every round. In a final round, the VM is halted only for the transfer of the last round of memory deltas.

VM Live Migration

The evaluation of Clark et al.'s work on a highly-utilized web-server resulted in a downtime of 165ms [26]. The link latency between the source and the destination has not been specified, but, as the authors have conducted the evaluation in an intra-datacenter environment, it can be assumed as being very small. Further VM live migration mechanisms have evolved since then [1], for example deduplication [159] and delta compression methods [154], resulting in downtimes between 200*ms* and 3*s*, if downtime was measured in the evaluation, at all. There have also been attempts to migrate VMs over WAN links [137, 159], however resulting in a sub-second latency which cannot be considered as seamless according to our definition in Section 2.7.1.1. A recent survey confirms the lack of reseach on VM migration in WAN scenarios [172]. For the time between delta rounds to converge, it is required that the bandwidth of memory changes (excluding multiple overwrites of the same page) does not exceed the network capacity to transfer the deltas. This condition is not always fulfilled with memory-buffered packet I/O (Section 4.4.2).

3.1.2 *Per-Flow-Based State Transfer*

Split/Merge

A problem which a transparent VM migration mechanism for NFV cannot address is *splitting or merging* of vNF instances. Here, an integrated state management usually requires the vNF at least to provide a classification of its state in order to correctly handle *shared* or *global* state among multiple or all vNF flows. The Split/Merge abstraction [140] avoids this problem by letting the NFV controller handle the associations of flows to vNFs, and defining forwarding rules for them in a fine-grained way. However, the fine-grained involvement of the controller can also be a drawback depending on the scale of flows, as it results in a possibly large overhead not only for the controller, but also for the forwarding elements being involved in the redirection of every particular flow during migration. The controller must also care about shared state, for example in an IDS, multiple flows handled by the vNF may be identified as a coordinated attack having a common state, and thus should be handled by a single node for best performance, if possible.

ClickOS

Works like ClickOS [106, 107] support the creation of thousands of tiny VMs or containers which can even be provided on a per-flow basis. Dietz et al. [32] show on the example of a virtual BRAS that this can be exploited for NFV state migration: By executing a VM migration on a per-flow basis, downtimes between 800*ms* and 1200*ms* can be achieved even with a VM *paused* during state transfer.

Chained state transfer

Olteanu et al. [128] use ClickOS to implement a carrier-grade NAT virtual middlebox. The authors chose NAT over other functions because of its simplicity while maintaining a per-connection state which must be transferred. For state migration, a new NAT NF instance is created in the datapath behind the original one, thus the instances are *chained*, and flow states are transferred successively from the first instance in the chain to the new instance. The first NF continues operation, but only processes flows with a state not yet transferred, while passing flows with state already transferred to the new NF. Their solution avoids frequent, costly SDN controller interaction, as per-flow redirection is done by the source NF instance during migration. However, the chained NFs bear the risk of increasing jitter and latency, which could become unacceptable in WAN migration scenarios. Furthermore, the state migration mechanism has been especially designed for NAT and is not proposed as a generic abstraction.

3.1.3 *Snapshots with Deterministic-Replay-Based Resynchronization*

Deterministic Replay

Besides the aforementioned delta-based snapshot resynchronization which tracks internal memory changes, and the per-flow-based state transfer, which requires frequent SDN controller interaction, another group of snapshot-based mechanisms assumes *deterministic* behavior of the internal state change in an NF instance based on input events triggering the state change. Therefore, they capture information about *external events*, send them to the destination NF which needs to be resynchronized, and execute the events again so that the destination's state gets updated like at the source (they *replay* them).

H. Liu et al. [97] use *ReVirt* [36], an event capturing mechanism originally proposed for intrusion detection, to identify external events in a VM for deterministic-replay-based live migration. The aforementioned method is capable of logging all possible external events, however, in NFV, we can focus on the external events of *network packets*. *OpenNF* of Gember-Jacobson et al. [57, 58] enables split and merge functionality for snapshot-based state transfers for which it proposes a new state management interface where the NF classifies state as *per-flow*, *multi-flow* (shared among multiple flows) or *all-flow* (global) state. Unlike SplitMerge [140], OpenNF avoids frequent SDN controller interaction by using *flow spaces* to split the flows with a few SDN rules only. Because the state cannot be migrated in a per-flow way in this case, OpenNF proposes a deterministic-replay-based state resynchronization method. Unlike H. Liu et al., OpenNF focuses on *packets* as external events only. After taking the snapshot at the source VM, all incoming packets are buffered at the controller node, When the snapshot is installed and the destination is running, buffered packets are re-injected at the destination to update the snapshot to the most recent state, before switching the traffic over to it. We refer to this deterministic-replay-based method, which focuses on network packets, as **duplication-based** snapshot resynchronization.

OpenNF

OpenNF – Extensions

OpenNF has been shown to operate *loss-free* and can move NFs in between 100*ms* and 700*ms* in an intra-datacenter environment, however, the packet buffering during migration results in an additional latency between 50*ms* and 150*ms*. The problem of OpenNF in its original state is that the buffering occurs at the controller, creating a bottleneck. Therefore, the authors have extended it [56] by offloading dataplane processing from the controller. Instead, the source VM duplicates packets, which are then re-injected at the destination VM instance. With the improvements, the latency overhead improves to values between 5*ms* and 20*ms* for redirected packets during migration. The improved versions of OpenNF can be classified as a seamless migration mechanism regarding many real-time applications, but the authors' evaluation of OpenNF lacks migration results over WAN-latency links and over links with limited capacity. J. Liu et al. [98] propose an improvement of OpenNF where flows of duplicated packets are scheduled over different paths in a network to reach a larger utilization and avoid congestion caused by updates.

However, alternative paths might not always be present in networks, especially in WAN scenarios. L. Liu et al. [100] detect flows which are short-lived and will likely expire during or shortly after migration. These flows are therefore excluded from the OpenNF migration process, while only the very small fraction of long-lived flows are considered for migration. Some migrations for specific workloads in their evaluation are seamless according to our definition, but the evaluation lacks WAN migration conditions.

3.1.4 *Miscellaneous*

Stateless NFs

An alternative approach to avoid state migration overhead is to remove the state from the instances. Kablan et al. [82] use a centralized instance for state maintenance, a *random access memory (RAM) cloud*, where NF instances connect to via *Infiniband*. Thus, a migration destination instance can take over the RAM cloud connection with no need for state transfer. Their results show that the low-latency interconnection between RAMClouds does not impact round-trip times compared to local RAM state at the instances. While being a suitable alternative for state migration inside datacenters, the presence of low-latency interconnection links is required, which is not provided over WAN links.

The state management task is also an important topic for testing network policies in NFs [50]. The *Slick* framework of Anwer et al. [3] provides a programming environment to decompose an NF into subsets of traffic processing, which can be individually placed.

3.1.5 *NFV State Migration: Summary*

Current literature lacks an instance migration mechanism which can seamlessly transfer the state and redirect the traffic of an NF instance to a distant site if the available bandwidth for state transfer is only small compared to the NF workload. In particular, such a mechanism, which we propose in Chapter 4, fulfills the following properties:

- The mechanism keeps the *resynchronization traffic* relatively *small* compared to the dataplane workload and the link capacity. The current state of the art requires a bandwidth for resynchronization as high as the dataplane workload of the NF instance (Section 4.4) for resynchronization convergence during seamless state transfer.

- By keeping the resynchronization traffic small, the mechanism has been shown to not only operate *seamless*[1] over high-performance and low-latency links inside a datancenter [26, 57], but also over weaker, *low-capacity* and *high-latency WAN links*.

- Like OpenNF [56, 57] and VM Live Migration [26], the mechanism is a snapshot-based *instance migration* mechanism, it does not transfer the flows

1 According to our definition in Section 2.7.1.1.

handled by an NF instance one-by-one or in small groups [140]. It therefore only requires a few SDN controller interactions to switch over to the new instance, not stressing the SDN controller and the forwarding hardware through frequent interaction. Furthermore, it does not require a partitioning of the state into individual flows.

3.2 PROGRAMMABLE HARDWARE ACCELERATION IN NFV ENVIRONMENTS

Kachris et al. [83] have sketched the idea of FPGAs as a suitable platform for NFV, as they can be (re)-programmed to custom needs and provide a flexibility similar to commodity CPUs. With *partial reconfiguration*, FPGAs could even allow resource virtualization. The authors furthermore suggest an SDN-based internal network forwarding on the chip in order to distribute network traffic between the virtualized resources. Xilinx's *softly-defined networking* paradigm sketch of Brebner et al. [13] follows a similar idea. Here, FPGAs are embedded in an SDN forwarding logic. Match/Action rules of OpenFlow can be complemented by programs in a HDL which are executed on FPGAs if advanced functionality is required beyond OpenFlow capabilities. It is also possible to use a graphics processor to accelerate certain tasks of an NF [61].

Ge et al. [55] demonstrate a computing node capable of provisioning FPGA-based hardware acceleration resources to vNFs operating on it. The vNFs can choose from a pre-defined set of *accelerator* functions for their tasks, however, they cannot be freely programmed by the vNF. Furthermore, HWA resources are not pooled in an infrastructure, they must be provisioned on the same computing node. A provisioning of hardware acceleration restricted to the same node has also been proposed by Yamazaki et al. [168].

The provisioning of *pooled* FPGA resources in *cloud* infrastructures has already received some effort by the research community. While many works focus on the FPGA solving *computing* problems, Byma et al. [20] describe and demonstrate the dynamic allocation of resources on a network-attached FPGA to implement a *load balancer*. Similar to the proposal of Kachris et al., the authors have extended OpenStack by enabling the user to provision FPGAs and program separate regions on it with *partial reconfiguration*. The authors have tested how fast the FPGA resource can be provisioned, and in their work, the NF is either completely implemented on the commodity CPU or the FPGA.

At the publication time of our proposal in March 2015 [122], no transparent split-architecture for VMs had been described,

- which NFs can use to offload their simple and complex processing workloads to different hardware independently, in order to exploit the advantages of both the commodity CPU and the FPGA,
- which *pools* both resource types independently in the infrastructure for better elasticity, and provisions them over the network,

- which describes interfaces for traffic and state management of the aforementioned seamless provisioning and configuration.

After the publication of our concept which we describe in Section 5, an architecture following our idea has been proposed by Bronstein et al. [19], in which NFV hardware acceleration resources can be elastically provisioned and used in the context of a split-architecture like in our proposal. However, the *pooling of HWA resources* in an entire infrastructure has not been described. These efforts have been followed by *OpenBox* [14, 15], which describes a partial offload of suitable workload processing to different types of OpenBox instances (OBIs), the use-case split is achieved with *header classifiers*. Zhang et al. [173] have recently described the execution of network functions on heterogeneous FPGA and commodity CPU resources which can be provisioned depending on current performance demand, while their system misses a split-architecture for partial offloading of tasks which are too complex for an efficient execution on an FPGA.

Since 2016, standardization of accelerator provisioning interfaces has been driven by the ETSI [45, 46]. These recent works in research and industry underline the relevance of elastic hardware accelerator provisioning as envisioned in our initial proposal.

3.2.1 *Hardware Acceleration Elasticity Evaluation*

A metric to benchmark elasticity has been proposed by Herbst et al. [66, 67], which was accepted as a reference by the Standard Performance Evaluation Corporation (SPEC). The metric can be used to benchmark the adaptability of a service implementation to actual service demands, given that a pool of virtualized hardware with *homogeneous* performance properties and costs is available for flexible assignment. In their approach, a resource *demand* is determined and is set besides a *supply* of resources. The difference of supply and demand is the *accuracy* component of the benchmark, which captures over- and underprovisioning, both negatively impacting the benchmark score. Furthermore, the *timeshare* component considers the time a system was in a provisioned state. The metric *elastic speedup* can be used to compare an elastic system to a *baseline* system, which can either be another elastic system, or a system with statically assigned resources. However, the elasticity benchmark of Herbst et al. is not suitable for elastic provisioning of resources with heterogeneous performance properties and costs. Furthermore, the metric is not suitable to derive the costs which the operator can save through elastic provisioning.

Fahmy et al. [49] conduct a cost evaluation of providing FPGAs in the cloud for *computing tasks*. They investigate the performance of a Boyer-Moore [12] searcher implemented on both an FPGA and a commodity CPU. Based on the results, they conclude that an FPGA is more cost-efficient if it is 12% occupied or more, however only regarding energy costs (computational efficiency). The granularity of the partial reconfiguration regions which they use in their proof of concept (64 regions) is likely not achievable in NFV, as the remaining gate logic per region would be quickly insufficient to operate the logic elements of the NF instances themselves or the required distribution logic for high-performance Ethernet traffic between all these instances and the physical ports. For example, Byma et al. use 4 partial reconfiguration regions only [20] which we also assume in our evaluation. The lack of a model in current literature to describe the benefits of elastic hardware accelerator provisioning for network tasks in NFV is addressed by our contributions in Section 5.3.

3.3 NFS ON BARE-METAL SWITCHES

In contrast to OpenFlow-based dataplanes [108] which have been subject to lots of research, the bare-metal switch community is a very young research field. The likely reason is that work on bare-metal programmable dataplanes to implement arbitrary NFs bears the risk to end up in vendor-specific designs, which should be avoided in NFV infrastructures. However, given the very high throughput and latency guarantees of a bare-metal switch to a relatively low price, the topic has a high practical relevance for network operators (for example in the CORD infrastructure [27, 129]), and should therefore not be ignored by the research community.

Wang et al. [162] introduce *MiniReal*, an abstraction which virtualizes (slices) the resources of a bare-metal switch for multiple tenants. The authors propose the MiniReal testbed as an alternative to software-based *Mininet* installations. Another work of Wang et al. [163] concerns implementing software flow counters in bare-metal switches. Fritzsche et al. present Basebox [53], a controller for bare-metal switches. Their switches run the OF-DPA drivers together with an OpenFlow agent. With the agent, the switches connect to an OpenFlow controller which translates Linux netlink commands into OF-DPA rulesets, transferred via OpenFlow. Kurtz et al. [92] compare bare-metal switching to fully-virtualized switching when using it in 5G networks. For their comparison, they use Pica8 switches with PicOS, providing full OpenFlow support.

Thus, a few research works have used bare-metal switches in the past, however, they all use them via OpenFlow, either by encapsulating OF-DPA in OpenFlow, which results in harsh OpenFlow table type restrictions [53, 162] and likely reduces performance (for example through reaction delays), or by using a proprietary OpenFlow middleware [92], increasing costs. Furthermore, unlike our contribution described in Section 6, previous works are focused on forwarding and do not target to implement an NF using inherent features of a bare-metal switch.

4 SLIM - SEAMLESS INSTANCE MIGRATION

The ideas and findings presented in this section have been previously published by the author of this thesis [126, 127].

Chapter overview

In this section, we present SliM (Contribution 1.1) and evaluate its advantages compared to the state of the art (Contribution 1.2). SliM is based on the *statelet method* and interface which we motivate and describe in Section 4.2. In Section 4.3, we describe the SliM system model and the migration process. Based on the specified model, we conduct an *analysis* of SliM in Section 4.4, which also facilitates a comparison to the state of the art. In Section 4.5, we describe our implementation of SliM as an add-on for DPDK [23, 35]. Our implementation has been used for evaluation in a testbed, the setup is described in Section 4.6 and the evaluation results are discussed in Section 4.7. Before starting with the aforementioned sections, we conduct an analysis of the seamless migration problem regarding low-performance WAN links in NFV.

4.1 PROBLEM ANALYSIS

Focus of previous work

The state of the art has been mainly evaluated in testbeds with low-delay and high-throughput links [56, 57] Both delta-based VM live migration [26] and duplication-based state resynchronization [57] require a large amount of network bandwidth from a source to a target vNF during snapshot transfer and resynchronization phase. In high-performance networks, for example inside a datacenter, snapshot and resynchronization bandwidth can be easily provisioned, however, this does not apply to wide-area networks.

Problem description

Figure 6 illustrates the problem with an example of a vNF instance being migrated between two DCs. Here, a vNF instance is about to be migrated from a source DC (1), e.g. an edge cloud, to a target DC (2), e.g. at a central location. The vNF instance processes traffic between two endpoints, e.g. subscribers (3) connected to the edge cloud, and web servers (4) connected to the target DC. The data plane traffic is bi-directional: Upstream traffic (5) from endpoint A is processed by the vNF instance currently in the source DC (6), the processed traffic (7) is then sent over to the target DC over a symmetric, bandwidth-restricted WAN link with significant latency (8), where it is relayed to Endpoint B (4). In the other direction, downstream traffic (9) from Endpoint B is sent via the target DC to the source DC (over the WAN link), where it is processed by the source vNF instance and sent (10) to endpoint A. Additionally to the bi-directional traffic of the vNF, state transfer traffic is required to be sent from the source to the target DC (11). In this simplified model used to motivate the problem, bandwidth on the WAN link is required for the downstream

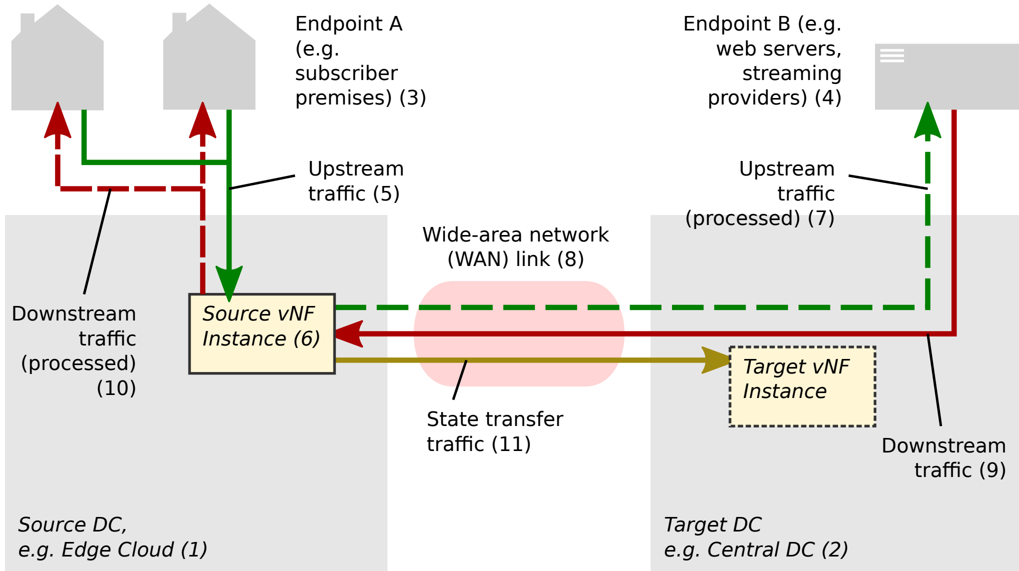


Figure 6: A common scenario in which state transfer traffic may become a bottleneck in instance migration.

traffic in the direction Target-DC -> Source DC, as well as for the upstream traffic *and the state transfer traffic*[1] in the direction Source DC -> Target DC.

WAN state transfer bottleneck

Remaining at our model in Figure 6, if we assume that the upstream and downstream traffic processed by the vNF instance needs to utilize the WAN link to a very large fraction (e.g. in a carrier scenario), the available bandwidth for the upstream traffic will be significantly reduced by the required state transfer bandwidth. Duplication-based mechanisms create copies of the packets for state resynchronization, and thus the required state synchronization bandwidth is equal to the combined upstream and downstream bandwidth[2]. This requires that the upstream and downstream directions can each only occupy below 33% (assuming downstream = upstream) of the WAN link's bandwidth during state migration, as more than 66% of the bandwidth is required for state synchronization traffic. *If the data plane occupies 33% or more of available bandwidth in each direction, the resynchronization traffic cannot keep up (migration fails), or packets on the data plane must be dropped which breaks the seamless behavior.*

Migration might fail...

Even if the WAN link provides sufficient bandwidth for data plane traffic and the state transfer, migration time will increase with less headroom for resynchronization and will approach infinity as the data plane traffic reaches 33% of the link capacity in each direction, because the snapshot resynchronization phase will take longer accordingly, having less bandwidth for keeping up with data plane packets.

1 Like in the analysis, we ignore signalling traffic required for state transfer, which is bi-directional (e.g. TCP acknowledgements).

2 Copying downstream packets at the target DC instead of the source would reduce the bandwidth requirements, but would lead to out-of-order packets and thus possibly break correct deterministic replay.

Finally, in duplication-based deterministic replay mechanisms [56] all incoming packets must be buffered until instance startup. Even if the WAN link has sufficient bandwidth, a high instance load bears the risk of overflowing packet buffers at the destination unless they are sufficiently sized, which would require large amounts of memory.

... like in delta-based mechanisms

For a seamless VM live migration [26], which copies invalidated memory as state *deltas* for resynchronization, it is required that the time for each successive delta round converges to a very small value, so that it is small enough to halt the VM for the last round without any perceivable service downtime. However, even only the memory activity caused by the *packet input* events may result in invalidated memory with a traffic volume as high as packet duplication.

Section 4.4 provides a more detailed and quantified analysis on bandwidth limitations, along with a more generic model, it also quantifies bandwidth requirements for VM live migration, which is more complex as it depends on the vNF's internal memory management.

4.2 APPROACH

Extension of Duplication

As mentioned in the introduction, our approach extends duplication-based state resynchronization. In OpenNF [56, 57], the packet duplication (at the source) and double-processing (at the destination) is a separate process from the data plane operation which is still at the source, thus this process does not have a negative impact on the data plane latency. Like OpenNF, our approach first sends a snapshot, and then focuses on *packets* as external events, which are buffered and deterministically replayed at the destination, in order to resynchronize the state of the snapshot, which is assumed to be outdated as soon as it has been fully received by the destination.

Dupes contain redundant information

In deterministic replay, network functions are assumed to change their state based on packets. However, most network functions require only a small part of the overall information in a packet to update their internal state [14]. Many NFs create or update a state only based on information on a few header fields in the packet, especially payload can be considered as redundant information regarding a state update. Our approach therefore focuses on sending a byte vector of only the required information for a state update, instead of the entire packet. The idea is to thereby reduce the state synchronization traffic, which we have identified as the bottleneck of the state of the art before.

> **Definition 1.1:** Conceptually, a *statelet* of a packet processed by an NF is the information in the packet which is required to obtain the state of the NF after the packet has been processed, given the state of the NF before the packet has been processed.

> **Definition 1.2:** From a software development view, a *statelet* is a variable-length byte vector containing the information as of Definition 1.1. The format of the statelet is NF-specific and transparent to any other component. The NF process

is capable of announcing a statelet for every ingress packet, and, given a statelet, the process can update its internal state as if the corresponding packet had been processed.

Listing 4.1: Simplified statelet interface provided to the NF in a C-like language.

```
// (1) Returns whether the NF shall currently
//announce statelets. Returns > 0, if yes
extern int shall_announce_statelet();

// (2) Called by the NF to announce a
// statelet at the source instance.
extern int announce_statelet(void* statelet, size_t len);

// (3) To be implemented by the NF to install
// statelets which have been received at the
// destination instance.
typedef int (*installStateletPtr)(void* statelet, size_t len);
extern int register_statelet_callback(installStateletPtr fPtr);
```

Statelet interface

The remaining question is how to determine which information in a packet is relevant. Our approach is to involve the NF in this task, which describes this per-packet information in a compact byte-vector representation, which we refer to as a **statelet** (Definition 1.1 and 1.2). Therefore, we propose an interface (Listing 4.1) which can be used by the NF and which is implemented by the execution environment (e.g. the hypervisor) of the NF. When the NF receives an incoming packet for processing, the NF checks whether it shall currently announce statelets with (1). If (1) requests that statelets shall be announced, the NF announces the statelet via a pointer in (2). The NF also receives statelets in order to install it. In the simplified example, the NF provides a pointer to an implemented installation function at startup, which is called by the environment during migration (3).

Thread safety

The NF interface is thread-safe: Statelet announcement (1) can be called by multiple threads simultaneously, thus, the NF does not require thread synchronization, for example if packets are received by multiple CPU threads (see Section 4.5 for details). Furthermore, statelet installation (3) is guaranteed to be triggered synchronously by the environment. For statelet announcements of the same thread on the source instance, the statelet installation (3) is furthermore guaranteed to be triggered in order. Finally, packet-in events are guaranteed not to interfere with statelet installation at the destination instance.

Types of statelets

The format of the statelet is NF-specific, the knowledge about the format is only relevant to the NF process. The statelet should be created as small as possible, however, it should be efficient to announce and install it, thus a trade-off must be found. An *empty* statelet (with a length of 0 bytes) might be created in case that only the information is required that a packet has entered (e.g. to increase packet counters). In case that no state change has occurred, no information is required about incoming packets, thus the statelet creation can also be *omitted*.

Deterministic execution, timing

In our approach, we have assumed *deterministic behavior* of the NF based on incoming packets. This means, the new state of the NF is defined by the previous

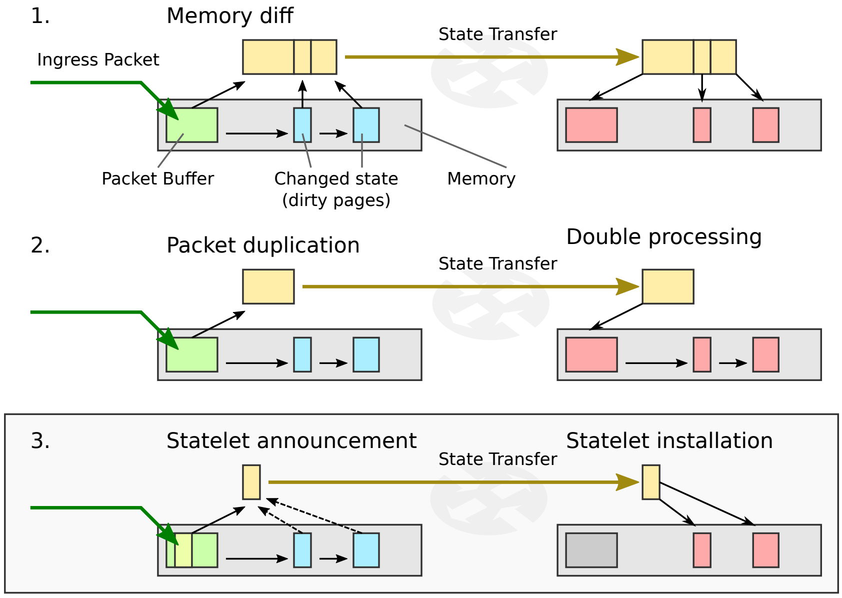


Figure 7: Comparison of different approaches for state synchronization. 1. VM live migration [26], 2. OpenNF with peer-to-peer transfer [56], 3. Statelet-based transfer (our approach). [127]

state and the information in the incoming packet only. Any randomness required (e.g. for cryptography) must be created by pseudo-random number generators. Another parameter influencing the state in NFs is the *time* when a packet enters the NF instance, which might be required to calculate rates and packet inter-arrival times. Packet timing is not considered by the proposed statelet approach (it has also not been considered in previous work [56, 57]). The basic statelet interface does *not* guarantee that a statelet is installed at the same time as the corresponding packet has been received, or that the statelet interarrival times are equal to the packet interarrival times. The solution is to consider the *timestamp* of the ingress packet, the time when it has entered the NF instance, as part of the statelet, whenever timing information is required. However, this requires a slightly larger integration effort, as it requires the NF process, for both packet processing and statelet installation, to consider the timestamps only, and to avoid calling the system time.

4.2.1 *Formal Description*

Statelets, formal description

A statelet can be formally described. We consider the current state of a vNF instance $s \in S$, as well as all information in a packet $p \in P$. We define the next state $s' \in S$ with Equation 1. Here, $f(s, p)$ is the state modification part of the network function process executed on an ingress packet event.

$$s' = f(s, p) \tag{1}$$

In order to use the statelet approach, the NF must provide a function $g : S \times P \rightarrow L \cup \{\diamond\}$ which returns a statelet $l \in L$ or omits statelet announcement ($\diamond$). Furthermore, the NF must provide a function $h : S \times L \rightarrow S$ for statelet installation, so that Equation 2 holds.

$$\begin{aligned} \forall s \in S, p \in P : \\ (g(s,p) = \diamond \vee f(s,p) = h(s, g(s,p))) \\ \wedge \\ (g(s,p) = \diamond \rightarrow f(s,p) = s)\,. \end{aligned} \tag{2}$$

In Equation 2, the first part states that either the statelet announcement has been omitted, or that the state after the processing of any packet p with any original state s (s-state instance) equals the state after installation of a statelet at an instance with the same original state, while this statelet has been previously announced by an instance after processing of packet p, also with the original state s. The second part requires that if statelet announcement is omitted, the NF instance's state is unchanged.

Figure 7 compares the concept of statelet-based resynchronization (3.) to memory-delta-based VM live migration (1.) [26] and duplication, also known as packet-based deterministic replay (2.) [56, 57]. In summary, our hypothesis is that the proposed statelet approach saves bandwidth occupied by the state resynchronization phase, as it does not transfer information redundant for the state update.

4.2.2 *Examples of Statelets for Popular NFs*

The statelet factor σ

In this section, we provide examples on the statelets of popular NFs and how to announce and install them. We furthermore argue about their expected size, which is NF- and packet-dependent. Based on a given example workload under realistic conditions, we calculate the **statelet factor** σ, which is defined as the average ratio of the statelet size compared to the size of the corresponding packet. The statelet factor is relevant for our analysis in Section 4.4.

4.2.2.1 *Network Address Translation*

NAT example

The term *network address translation (NAT)* refers to replacing source and/or destination IP addresses with others during processing. Usually, for returning packets, the translation is reversed, so that IP connections between endpoints can be established where at least one communication partner's address has been intermediately changed by the NF. In *static* NAT, the translation is fixed, so such an NF would operate stateless. Commonly, the term NAT is used for *masquerading*, a way to dynamically apply NAT and port address translations to individual TCP/IP or UDP/IP sockets.

Masquerading is well-known from home and business routers, but it has also been used by carriers in the last years because of the scarcity of IPv4 addresses. Here, it is commonly used to enable Internet communication for networks out of the *private IPv4 range* (which are not routed on the Internet), by letting the hosts on these networks share one or a pool of a few public IPv4 addresses. To achieve this, packets sent by a host in the private network to the Internet are processed by the NF: the source IP address and the source TCP/UDP port are replaced by a public address and a new port before being forwarded to the Internet.

The corresponding IP and port address translation must be remembered. The only information relevant for NF state change in the packet is the source IP address and port, the destination IP address and port and that it is a new, initial connection (a 2-byte ID). Thus, a statelet requires only **14 bytes** for initial connections. As the NAT function behaves as an Ethernet-to-Ethernet router in our example, it sends ARP requests, and inserts entries into an ARP table upon receiving ARP responses, thus, the ARP information (an IP address, a MAC address, and a 2-byte type identifier) must also be considered as a statelet with **12 bytes**. Finally, the example NF maintains packet and byte counters *per internal host* for statistics. Thus, the packet length is required (2 bytes), the internal IP address (4 bytes) and a 2-byte type identifier, resulting in a total of 8 bytes. The statelet factor for a workload of 512-byte packets, where every 15th packet is a new socket and no ARP responses are received, is $\sigma = 0,0167$.

4.2.2.2 *Intrusion Detection*

IDS example

An IDS scans network traffic for suspected malicious activity. While some activity can directly be detected with information in a single packet (e.g. signature-based), other activity, like coordinated attacks from distributed clients (e.g. Secure Shell (SSH) scans), can only be detected by maintaining a **state**. Intrusion detection is an example for a network function with global state which is difficult to be split and merged, and which cannot be reliably migrated per-flow (flow-based migration, e.g. [140]).

In our example, connection information in the header of every packet is tracked, as they might be relevant to identify future attacks, which we assume are **20 bytes** each. If a suspicious packet is received, it is recorded, resulting in the whole packet (512 bytes) to be relevant as a statelet. In our example, we assume 5% of the packets are suspicious, which results in a statelet factor of $\sigma \approx 0,087$.

4.2.2.3 *VPN Session Traffic*

VPN example

An endpoint of a VPN can initiate and tear down VPN sessions. In our example NF, we focus on the session phase, during which it provides VPN encapsulation/decapsulation and encryption/decryption. In our example, the statelet requires a **4-byte** identifier for the VPN session the incoming packet is belonging to. Furthermore, the VPN session works in Advanced Encryption Standard (AES)-CBC (Cipher Block Chaining) mode with a block size of 256 bits, the cipher requires the last 32 bytes of a packet's ciphertext (the packet's last cipher block) to be carried with the statelet. To maintain the order of packets, the statelet also needs to carry a sequence number of an ingress packet having 4 bytes. The example results in a statelet factor of $\sigma \approx 0,078$.

4.2.2.4 *Mobile Packet Gateways*

Mobile gateway examples

The statelet method could also be applied to various functions in mobile networks. For example in LTE's Evolved Packet Core (EPC), it could be applied to enable the Serving Gateway (S-GW) functions to seamlessly migrate between central DCs or DCs at the edge. For example, the S-GW provides packet data access to the Evolved NodeB (eNodeB) via the `S1-U`[3] interface [42]. Packets destined to the eNodeB must be encapsulated using the IP/UDP-based *General Packet Radio Service (GPRS) Tunneling Protocol Userplane (GTP-U)*, or decapsulated in the other direction. State management comprises (1) adding and removing sessions and bearers coordinated by the *Mobility Management Entity (MME)*, (2) currently associated mobile devices which change their eNodeB during handovers S-GW, (3) the current state of a handover between different S-GWs (X2- or S1-based), (4) the presence and the replication target for lawful interception, (5) packet and byte counters for accounting and billing.

Although the state structure is more complex, the state changes on incoming packets are similar to the operations in the NAT example, they comprise table-based state modifications (1-4) and counters (5), which are suitable for the statelet interface and commonly result in a small statelet factor. The example, however, emphasizes that a larger effort is necessary to integrate the statelet interface into *existing* implementations of complex NFs.

4.2.3 *Statelet Sizes and Packet Sizes*

Statelets are never larger than their packets

Statelets never exceed the size of their packet (plus its metadata[4]). Because of the assumption that a NF instance only changes its state deterministically based on incoming packets, a replay at the destination would never need more information than in the original packet. In the case that the information cannot be efficiently obtained, the method can always fall back to duplication of the entire packet. Thus, the bandwidth required by duplication-based state resynchronization constitutes an upper bound for the statelet approach. For the previous example NFs and workloads with a statele factor of less than 0.1, the state resynchronization traffic is only a small fraction of the data plane traffic. In Section 4.4, we will detail this and quantify the impact on migration performance.

4.2.4 *Performance requirements*

With statelet announcement and statelet installation, the statelet approach introduces two new operations, which might bear the risk that their invocation deteriorates data plane performance and makes the statelet approach impractical. Therefore, we state performance requirements on the implementation of the statelet interface by the execution environment, and which are guaranteed to the NF execu-

3 S1-Userplane; Userplane is a commonly used synonym for *Data Plane*.
4 Metadata would be timing information (e.g. a timestamp) and information about on which ingress port it has been received.

tion process. In Section 4.3 and 4.5, we will explain how the execution environment fulfills these requirements.

Low performance penalty

First, the statelet announcement call made from the data plane is supposed to return as soon as possible. If the entire statelet annoucement processing was carried out synchronously by the data plane thread, it would increase the per-packet processing time and thus would reduce the data plane performance of the NF instance. For example, like in our implementation (Section 4.5), the data plane thread might place the statelet pointer in a buffer and immediately return, while other threads might care about subsequent statelet processing. Secondly, statelet announcement is only required during a short period of time. The interface in Listing 4.1 provides a function (1) returning whether the NF shall announce a statelet. When an ingress packet is received, the NF process quickly checks the return code of this function and can avoid any statelet announcement overhead.

No data plane interference

Thirdly, during statelet installation, no data plane workload is required to be processed by the destination NF, the statelet installation can use the entire processing performance of the data plane. For example, if processing cores have been reserved in a VM for the vNF data plane process, they are free for processing the statelet installation. Thus, the per-statelet installation time on a processor can be as high as its per-packet processing time.

4.2.5 *Interface integration effort*

Integration effort

Compared to many other mechanisms for instance migration [1, 26, 57], mechanisms using the statelet approach have the drawback that they are *not transparent* to the NF. The statelet approach requires the NF developer to adapt the NF code to use the statelet interface, which might result in a larger integration effort. However, state management required for enabling split and merge operations of NF instances also requires novel interfaces used by the NF. For example, OpenNF [57] is transparent regarding state resynchronization, however, it demands that state is classified whether it is bound to only a particular flow, a group of flows, or the entire NF. A future integration effort can be reduced by combining the interfaces which have been suggested to improve different aspects of NFs in a single, handy API.

Code analysis

We assume that the aforementioned integration effort might be avoided in future work by applying code analysis techniques together with an automated refactoring in order to implement the statelet interface. For example, Khalid et al. [85] suggest and demonstrate analysis techniques for the aforementioned state classification required by OpenNF [57] to do splits and merges. To automatically generate code which returns the statelet of a packet, a possible approach would be to track the code's modifications in the memory which is persistent across multiple packet modifications, and which parts of the packet have been read that directly or indirectly influence these modifications – these parts constitute the statelet. Likewise, the statelet installation code could be automatically obtained by taking the data plane code and only keeping the code making the aforementioned modifications in persistent memory. Code analysis techniques to automatically implement the statelet interface are not part of this thesis, and are left for future work.

4.3 SLIM SYSTEM MODEL

This section describes **SliM**, a method for *seamless instance migration* in NFV infrastructures. Like VM live migration [26] or OpenNF [57], SliM transfers a snapshot of the current state. When the state has been transferred to the destination, it is assumed to be outdated, thus, SliM then uses deterministic replay to resynchronize the state at the destination. Instead of using external events [97] or duplicated packets [56, 57], SliM uses the statelet method discussed in Section 4.2, thus requires the NF implementation to announce and install statelets, if requested.

Section overview

In this section, we describe the model of the SliM system and detail the state migration process. In Section 4.4, the analysis of SliM is conducted based on this model. Furthermore, in Section 4.5, we describe an implementation of SliM which is based on the system model described in this section.

4.3.1 *Assumptions*

System dependencies

The system model is based on our architecture described in Section 2.6. In more detail (Figure 8), we assume multiple *computing nodes*, which can provision compute and memory resources to vNF instances. The computing nodes are managed by a logically centralized NFV controller, which shall not be involved in any data plane processing. The nodes can communicate with the controller and with *each other* over a virtual control network (for example a VLAN). Besides the control network, which always enables end-to-end communication between any computing nodes, connectivity of an NF instance's data plane to the network can be established over a flexible SDN which might be subject to significant latency and limited throughput *shared* with the control network. By request of the NFV controller, the SDN starts forwarding traffic from a traffic source to a virtual receive port (A_0 or A_1)[5] of a vNF instance, and from a virtual transmit port (B_0 or B_1) of the instance to the intended destination. The controller is able to flexibly reassign these forwarding paths to other instances' virtual transmit and receive ports *separately*.

Local snapshots

In our model, the vNF instance environment (VIE) can take consistent local snapshots of the VM's state. The snapshot process may take time, and the snapshot might be outdated after finishing the snapshotting process, but the snapshot must exactly have the state at the specific point in time when it was triggered. Furthermore, there must not be any interruption of the data plane process that results in a perceivable jitter or packet loss. Taking a consistent snapshot under these conditions is challenging, as the state will continuously change during data plane operation.

Various strategies are available for taking the initial state snapshot. For example, we can capture a serialized representation of the current VM's memory, which does not require implementing snapshotting routines at the NF. Alternatively, the state can be actively serialized by the vNF code itself, and a new vNF instance can recover from this serialized state representation. The latter approach is not transparent, but allows to only select relevant parts of the NF memory state. The problem of

5 In this example, the NF has 2 bi-directional interfaces.

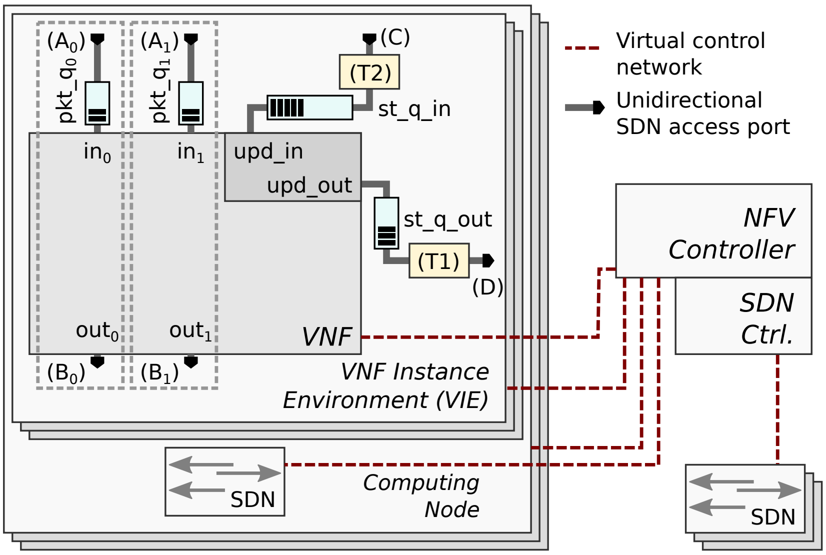


Figure 8: Architecture overview, 2 full-duplex interfaces. [127] (Adapted)

snapshot consistency can be tackled by *isolated snapshots* or *redirect-on-write*, we provide a light-weight implementation in Section 4.5.2.

4.3.2 *Definitions and Interactions*

Partial migration

The migration is always conducted from a *source* instance (`srcInst`) to a *destination* instance (`dstInst`). To scale out with splits, we require **partial** state migration, which transfers only a part of the state to a destination, while leaving the remaining state in operation at the source. The SliM mechanism supports partial migration for seamless splits and merges, the difference to the default SliM process is being discussed in Section 4.3.5.

The vNF instance environment (VIE)

The SliM model (Figure 8) introduces a VIE in which the vNF is embedded, and which manages the migration process locally. The VIE can directly communicate with the NFV controller. The VIE provides the NF with one or multiple full-duplex network interfaces (dashed, light-gray rectangles). Packets arriving from the SDN data plane (A_0 or A_1) are *buffered* in a packet queue `pkt_q` for each interface, while packets leaving the vNF instance are directly handed over to the SDN data plane[6].

The statelet datapath

Unlike common virtualization environments, the VIE now also provides the statelet interface to the vNF (Figure 8, dark-gray box in the upper left corner of the vNF). In this abstraction, the statelet interface is depicted as two statelet pipes `upd_in` (statelet installation) and `upd_out` (statelet announcement). Statelets announced via `upd_out` are buffered in `st_q_out` and are encapsulated in a transmission process ensuring reliability and flow control (T1). Statelets received are decapsulated (T2) and queued in *st_q_in*. By request of the process described in

6 Traffic sent through the transmit interface might be buffered by transmit queues of the system's network interface

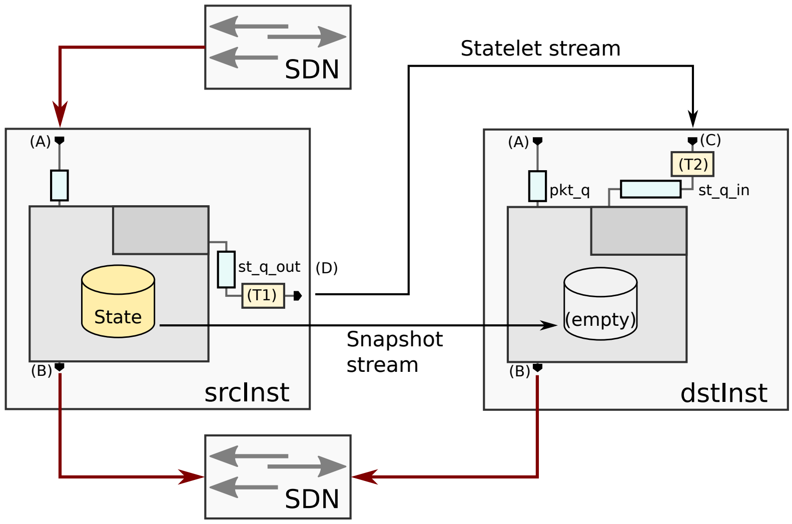


Figure 9: Datapath before redirection of the traffic. [127]

Section 4.3.3, threads picking up packets from the aforementioned queues may be individually *blocked* for a short amount of time, so that their service rate is *zero*.

4.3.3 *Migration Process*

Triggering process execution

The migration process described in the following is prepared and triggered by the NFV controller (however, during migration, source and destination vNF instance directly communicate with each other). The controller receives the request for migrating a vNF from an external component, together with the respective source NF instance and the destination computing node. This external component, deciding about when and where to migrate a NF, might be a dynamic placement process, which is out of this work's scope, but has been investigated in previous work.

Initial state

Figure 9 illustrates the SliM migration process *before* switchover of the data plane. Initially, `dstInst` is not active. `srcInst` processes data plane traffic, thus the SDN datapath from the traffic source is connected to `srcInst`'s ingress port (A), and from `srcInst`'s egress port (B) to the traffic destination (dark red arrows). If the vNF instance comprises multiple interfaces, the SDN is supposed to have created paths for every interface separately. The statelet interface is unused, as the VIE does not request statelet announcement.

4.3.3.1 *Stateless destination instance preparation*

An "empty" VM is booted

Upon the request to migrate the vNF instance to a specific destination, the vNF process and the VIE is booted up at the destination with an empty state and an inactive data plane. The SDN does not yet forward ingress traffic to the data plane receive port (A) at `dstInst`. However, the SDN already prepares a path for egress traffic from the data plane transmit port (B) to the destination, which is used in the later steps. At `dstInst`, the queues `pkt_q` does not have an active data plane process dequeuing packets, and `st_q_in` do not have a statelet dequeueing process yet, thus, the queues are blocked.

4.3.3.2 *Snapshot and statelet transmission at the source instance*

Streams

After `dstInst` has correctly booted up, `srcInst` connects to dstInst using a reliable, stream-based transmission protocol with flow control capabilities. Layer 4 connections (for example TCP) over a special SDN path or over the virtual control network might be used to implement connectivity, the streams are supposed to be *opened* and *closed* by `srcInst`, which is also used for signaling to the destination. While connecting, `srcInst` opens two streams to dstInst, a *snapshot* stream and a *statelet* stream, and hands each of them over to a separate thread:

Snapshot thread

The **snapshot thread** first sets the *shall_announce_statelet* flag, so that the vNF from now on asynchronously announces statelets of packets entering the vNF instance, and enqueues them in `st_q_out`. It then immediately triggers the snapshotting process and sends the serialized snapshot over the snapshot stream to the destination instance. When the snapshot has been fully serialized and transferred over the stream, the stream is closed and the thread terminates, but the *shall_announce_statelet* flag remains active.

Statelet dequeue thread

Concurrently to the snapshot thread, the **statelet dequeue thread** waits for new statelets now continuously enqueued in `st_q_out` and sends them over the statelet stream to the destination instance (T1). The snapshot thread continues until the following steps explicitly interrupt it.

4.3.3.3 *Snapshot and statelet reception at the destination instance*

Upon receiving the connection attempts of `srcInst` to transfer the snapshots and statelets, `dstInst` also opens two separate threads to process the two concurrent streams.

Statelet enqueue thread

The **statelet enqueue thread** (T2) receives statelets from the statelet stream and enqueues them in `st_q_in` (which is initially blocked and starts to fill up). Like the statelet dequeue thread, it continues until the following steps explicitly interrupt it.

Resync thread: Snapshot installation

The **resynchronization thread** receives the stream with the serialized snapshot and installs the snapshot at `dstInst`. As `pkt_q` and `st_q_in` is blocked, it is guaranteed that state modifications due to snapshot installation are not disturbed by concurrent data plane operations or updates due to statelet installation.

Resync thread: Statelet installation

After the snapshot has been installed and the snapshot stream has been closed by the source, the resynchronization thread starts to dequeue statelets from `st_q_in` and installs them via *synchronously* calling the statelet installation routine provided by the vNF process via the statelet interface. During statelet dequeueing, the resynchronization thread continuously checks the number of elements in `st_q_in`. If the number of elements is lower than a specified threshold d, it sends a *switchover request* message to the SDN controller, but continues its dequeueing and installation operation even after sending the message.

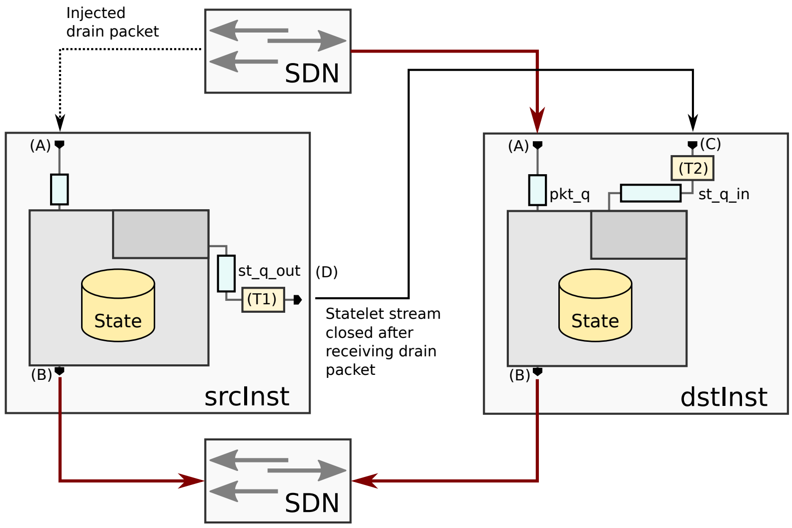


Figure 10: Datapath after the redirection of the traffic. [127]

4.3.3.4 *Drain-packet-signaled start at the destination instance*

Drain packet injection

After receiving the switchover request message from the destination instance, the NFV controller immediately instructs the SDN to redirect the path from the traffic source to the `dstInst`'s data plane interface (Figure 10, A). The last packet sent to the data plane interface at `srcInst` is a special **drain packet**, which is immediately sent during switchover, and which signals to `srcInst` that the last data plane packet has been received, and all subsequent packets are handled by `dstInst`.

When `srcInst` has received the drain packet on all ingress network interfaces, it immediately closes the statelet stream, as no more statelets need to be announced by `srcInst`.

Closing the statelet stream

The dequeue thread at `dstInst` stops its dequeueing operation when the statelet stream has been closed *and* `st_q_in` is empty. Before terminating, the statelet dequeue thread activates the data plane at `dstInst`, so that packets buffered in `pkt_q` are dequeued and the data plane starts operation. `srcInst` can now be shut down.

4.3.4 *Thread Safety and Consistency*

Thread safety guarantees

Assuming that a mechanism for isolated snapshots at `srcInst` is available, the aforementioned SliM process is thread-safe and guarantees thread safety to the vNF process, assuming the vNF process is thread-safe without SliM. Furthermore, it guarantees consistency through in-order statelet installation. The SliM process guarantees that

- during data plane operation, no snapshot and no statelets are installed,
- no statelets are installed during snapshot installation and vice versa,

- statelet installations are not executed concurrently,
- statelets announced in a specific order by a data plane thread at the source are installed in the same order at the destination.

4.3.5 *Partial State Migration in SliM*

Additional assumptions for partial migration

The aforementioned SliM state migration process (Section 4.3.3) assumes a *full migration*, a complete transfer of the state from a source to a destination instance. However, for splits and merges of vNF instances in order to conduct scale-out and scale-in operations, *partial* state migration and *state merging* is required. In this section, we describe modifications to the SliM procedure required for supporting partial state migration. Given an instance `srcInst` being split into `dstInst_1` and `dstInst_2`, or `srcInst_1` and `srcInst_2` are merged on `dstInst` respectively, we make further assumptions:

- The snapshot of the state at `srcInst` can be split. It is possible to serialize a snapshot with the state for `dstInst_2` only instead of serializing the entire state of `srcInst`.
- The arriving snapshot streams of `srcInst_2` and `srcInst_2` can be merged at `dstInst` (SliM guarantees thread safety and exclusive access to the state memory).
- Starting at the time of the snapshot, the NF process can determine whether an ingress packet will be handled by `dstInst_1` or by `dstInst_2`.

The aforementioned state management problems are not part of our contribution, but have been extensively investigated in previous work [57, 140].

4.3.5.1 *Handling vNF instance splits*

Split process

To conduct a vNF instance split with SliM, only a certain part of the state is transferred to a new instance `dstInst_2`, while the other part remains in operation at the source as `dstInst_1`. The following changes must be applied to the SliM process:

- The isolated snapshotting process only serializes state required for `dstInst_2`.
- Statelet announcement at the source is only made for packets changing the state at `dstInst_2`, e.g. packets later handled by `dstInst_2`.
- Upon receiving the switchover request message, the SDN controller only switches over traffic which shall be handled by `dstInst_2` to the new instance. Nevertheless, the drain packet is injected.
- Instead of stopping the data plane operation at the source after receiving the drain packet, it is continued for the remaining traffic handled by `dstInst_1`.

4.3.5.2 *Handling vNF instance merges*

Merge process

Unlike the split, a merge of the state of a running instance into another instance in operation is not intended by the SliM approach. The reason is that the snapshot installation and statelet resynchronization would interfere with a data plane in operation. This approach would not be inherently thread-safe, this could only be solved with expensive mutual exclusion routines like thread locking, probably deteriorating data plane performance. Therefore, for a merge, a third instance `dstInst` is booted up, serving as the destination for `srcInst_1` and `srcInst_2`. The following changes must be applied to the SliM process:

- A merge is triggered by simultaneously triggering a full migration at `srcInst_1` and `srcInst_2` to the same destination instance `dstInst`.
- As a result, two statelet streams are simultaneously dequeued to `st_q_in`.
- The resynchronization thread receives the first and the second snapshot over two snapshot streams and merges them to obtain the new state of dstInst, before continuing with dequeueing and installing the statelets from `st_q_in`.
- Upon receiving the switchover request message, the SDN redirects the data plane of `srcInst_1` and `srcInst_2` to the new instance, and injects drain packets into each of the original data plane paths.
- The resynchronization thread at `dstInst` stops the statelet installation and starts the data plane operation as soon as `st_q_in` has underrun and *both* statelet streams have been closed.

4.4 ANALYSIS

Section overview

In the beginning of Section 4, we have introduced the problem of current instance migration mechanisms in a network subject to limited WAN bandwidth and significant delay. In this section, we perform a quantitative analysis not only on the performance of state of the art mechanisms, but also on the SliM approach mentioned in Section 4.3. Section 4.4.1 not only quantifies the problem statement, it also compares the concept of duplication-based state resynchronization [56] with our approach, *based on a model*. Section 4.4.2 provides results on total migration time and downtime of delta-based state resynchronization mechanisms like VM live migration [1, 26], if the VM is subject to a high packet I/O load, which can be expected in NFV. These results are obtained by numeric calculation, based on an optimistic memory management model with packet ring buffers.

In addition to the results in this section obtained by *analysis*, we also provide results based a testbed evaluation in Section 4.7.

4.4.1 *Analysis of Packet- and Statelet-Based Deterministic Replay*

Parameter definitions

In this section, we analyze interdependencies between the following quantified properties:

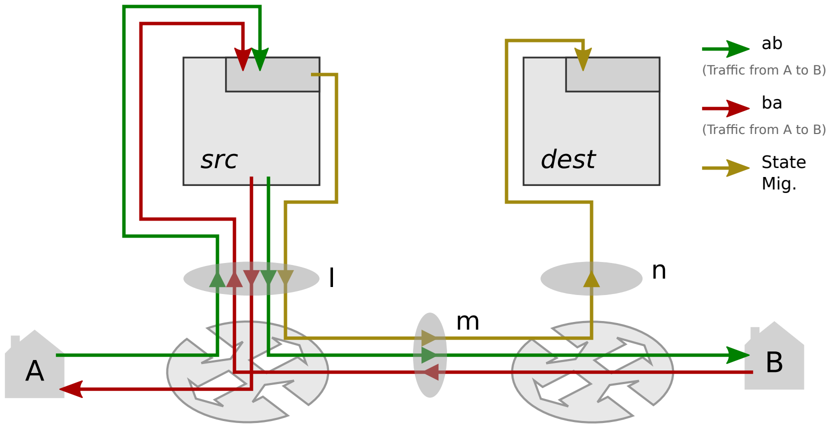


Figure 11: Physical and link layer model of an NFV infrastructure considered in the deterministic replay analysis. [127]

- **Statelet factor,** σ. The statelet factor has already been mentioned in 4.2 along with the examples of statelet announcement and installation for specific vNF types. It depends on the vNF type or implementation, and on a specific workload (a sequence of packets processed by the vNF). It is defined as the ratio of the average statelet size announced by the vNF when processing the workload, divided by the average packet size of this workload. For example, when the vNF processes a stream of packets with a size of 500 bytes and announces statelets with a size of 50 bytes for these packets, the resulting statelet factor is $\sigma = 0.1$. **A statelet factor of $\sigma = 1$ corresponds to duplication-based state resynchronization**, as the entire packet is transferred.
- **Snapshot size**. This is the size of the serialized representation of the state at the time of taking the snapshot at the source.
- **Link capacity**. In the model proposed in the following, the capacity of the links between the entities in the network are restricted to a certain value. The link capacity must be shared between data plane traffic and state migration traffic.
- **Data plane capacity**. The bandwidth available for the data plane traffic served by the vNF instance.
- **Seamless migration**. Based on the model, we can state whether the conditions, for example the link capacities, allow seamless migration or not.
- **Migration completion time**. If seamless migration is possible, this is the time from requesting the migration to the completion of the entire migration process, including snapshot transfer and resynchronization.

Analysis network model

Figure 11 illustrates the model which is used for analysis. The model describes the conditions in a network while migrating a vNF instance from a source to a destination, while processing bi-directional traffic between A and B. Furthermore, the model assumes three link bottlenecks: l, the shared path of $(A \Leftrightarrow sourceinstance)$ and $(B \Leftrightarrow sourceinstance)$, m, the shared path of $(A \Leftrightarrow B)$ and $(sourceinstance \Leftrightarrow destinationinstance)$, and n, the shared path of $(A \Leftrightarrow destinationinstance)$ and $(B \Leftrightarrow destinationinstance)$. Additionally to the data plane traffic between A and B flowing through the three bottlenecks and the instances, state migration traffic must be transported over the links (yellow arrow).

Model restrictions

The model focuses on throughput and bandwidth reservation, the influence of link latency is not considered by the model, especially the latency incurred by a WAN. The model assumes a constant data plane throughput. It also neglects traffic caused by control messages, including TCP overhead. It neglects flow control mechanisms like TCP slow start, the throughput is immediately available. The model assumes that no packet loss is caused by lower layers (no retransmissions necessary). The model provides a condition that the available bandwidth is never exceeded, however, it neglects that packet loss due to congestion can occur below this threshold, for example caused by microbursts or by small deviations from the steady state in conjunction with limited queue sizes. Finally, CPU processing speed and the bandwidth of internal interfaces in the hypervisor (for example the PCI bus) are assumed to be unlimited. Nevertheless, all the aforementioned parameters neglected by the model are taken into account in our testbed evaluation described in Section 4.6.

Bottleneck

In Figure 11, we have described three bottleneck links l, m, and n, their total maximum bandwidth capacity is defined by C_l, C_m and C_n respectively. F_{ab} or F_{ba} are defined as the current throughput of the datapath between A and B. Equation 3 defines C_{rem}, the remaining bandwidth available for state transfer traffic between source and the destination, which is the minimum of the remaining bandwidths on l, m, and n.

$$C_{rem} = \min\{C_l - F_{ab} - F_{ba}, C_m - F_{ab}, C_n\} \tag{3}$$

Condition for migration in finite time

The remaining bandwidth must be used for the snapshot transfer and the snapshot resynchronization phase. T_B is the combined local snapshotting time at the source and the installation time at the destination caused by CPU overhead, and S is size of the serialized snapshot. As we assume the snapshot to be of a finite size, we can assume that is is transferred in limited time over any available bandwidth larger than zero. For resynchronization, the source vNF instance currently processes packets with a throughput of $F_{ab} + F_{ba}$ (two receiving interfaces for both directions). Having the statelet factor, we can derive the bandwidth requirement for transferring the statelets with $\sigma \cdot (F_{ab} + F_{ba})$. Finally, we can derive the condition that the statelet stream (or, with $\sigma = 1$, the stream of duplicated packets) can be transferred over the remaining bandwidth in finite time without congestion, shown in Equation 4.

$$\sigma \cdot (F_{ab} + F_{ba}) < C_{rem} \tag{4}$$

Total migration time

If the aforementioned condition is not fulfilled, the statelet stream cannot be delivered to the destination instance, while ensuring that no packet loss must occur on the data plane at the same time. (Depending on the implementation, this will probably result in an overflow of `st_q_in`, packet loss on the data plane, or both.) If the aforementioned condition is fulfilled, we can obtain the total migration time with Equation 5.

$$T_{all} = T_B + \frac{S}{C_{rem} - \sigma \cdot (F_{ab} + F_{ba})} \tag{5}$$

Here, the statelet stream $\sigma \cdot (F_{ab} + F_{ba})$ is assumed to be a constant stream, which we have to subtract from C_{rem} to obtain the available bandwidth to transfer the snapshot with size S, given that statelets must not be buffered at the source in `pkt_q_in` and are always accurate at the destination. By dividing the snapshot by the remaining bandwidth and adding T_B, we obtain the time required to transfer the snapshot over the remaining bandwidth. We might now prioritize the snapshot stream, so that the snapshot is transferred faster, but the statelets must be buffered at the source in order to be sent after snapshot transmission. However, the sum of the traffic volume to be transferred is equal in any case (ignoring transmission *overhead*), and the data plane cannot start operation at the destination before both the snapshots and the statelets have arrived. Thus, T_all is the total migration time in any case, no matter whether we favor the statelet and the snapshot stream for transmission. By transforming the equation, we can obtain the minimum allowed statelet factor required to complete a migration in a specified time interval (Equation 6).

Statelet factor

$$\sigma < \frac{C_{rem} - \frac{S}{T_{all} - T_B}}{F_{ab} + F_{ba}} \tag{6}$$

Analysis of a 1Gbit/s link limit

In an example, we set $C_m = 1000 MBit/s$, resembling a WAN bottleneck at m. l and n are assumed to be high-bandwidth links thus we set $C_l = C_n = 10 Gbit/s$. The snapshot size of the vNF is assumed to be $20 MByte$, and the combined snapshotting time at the source and the destination (CPU overhead) is $0.3s$. Given these values and the aforementioned equations, Figure 12 depicts the interdependencies between data plane capacity, the maximum allowed statelet factor σ, and the total migration time T_{all}. Multiple curves are shown for different values of T_{all}.

In Figure 12, duplication corresponds to a statelet factor of $\sigma = 1$, and the vNF examples from Section 4.2 are located below $\sigma < 0.1$. We can see that the available data plane capacity is much higher when using statelets than when using duplication. For example for $T_{all} = 1.5s$, the data plane capacity available in each direction is at $\approx 800 Mbit/s$, while it is only $\approx 300 Mbit/s$ when using duplication. For very small statelets, data plane capacity can reach almost $900 Mbit/s$.

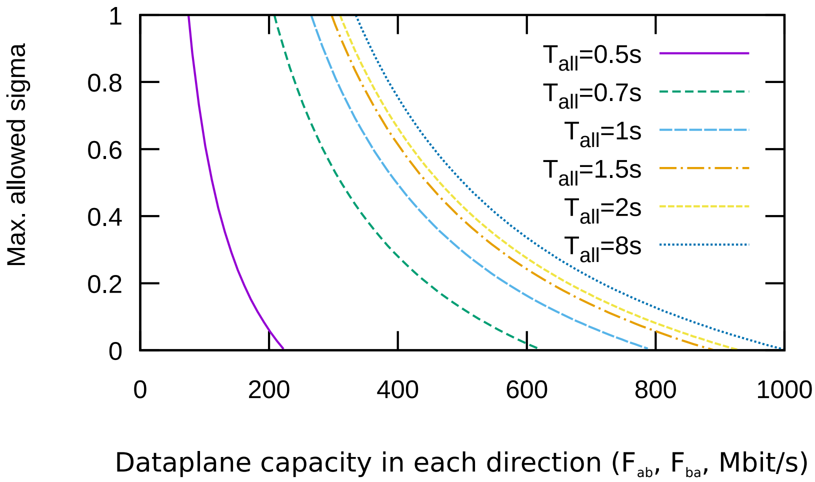


Figure 12: Determination of the highest possible statelet factor for a successful migration. Duplication corresponds to a statelet factor σ of 1. [127]

4.4.2 *Delta-Based VM Live Migration Under Network I/O Load*

Evaluation method

Delta-based state resynchronization used by VM live migration mechanisms [1, 26] have been evaluated by previous work to have downtimes which are not seamless according to our definition in Section 2.7.1.1, even in low-latency and high-bandwidth environments. The state resynchronization traffic depends on the amount of invalidated memory state during transferring and installing the last round. It is difficult to quantify the amount of invalidated memory as it depends on implementation details of the vNF and even the operating system. During I/O, network interfaces normally load the entire packets into memory, most high-performance implementations configure the interface cards to store the packets in a *ring buffer* or a similar buffer via DMA. A NF using the aforementioned ring-buffer-based packet I/O mechanism is subject to the memory invalidation caused by it, however, it might invalidate additional memory caused by its internal state management. Therefore, by modeling the invalidation of the ring buffer, we can calculcate a *lower bound* of the total migration time and the last-round downtime.

Ring buffer description

The ring buffer to be modeled consists of a memory region of a fixed size and a *head* and a *tail* pointer in this region. Initially, the pointers are on the same position. If a packet is stored in the buffer, it is written to the head pointer's offset and the head pointer is moved to the end of the packet. Likewise, a packet is fetched from the buffer from the tail pointer, and the tail pointer is moved to the start of the subsequent packet. If the pointer (or the read/write operation) would reach outside the memory region, we subtract the memory size from the address of these operations, thus, the operation *wraps around* (like in a ring) and continues at the beginning of the memory region.

Time of a delta transfer round

In this model, we assume r_{DP} to be the current data plane load and r_c the total link capacity. The size of the serialized snapshot is m_0, which can be considered as

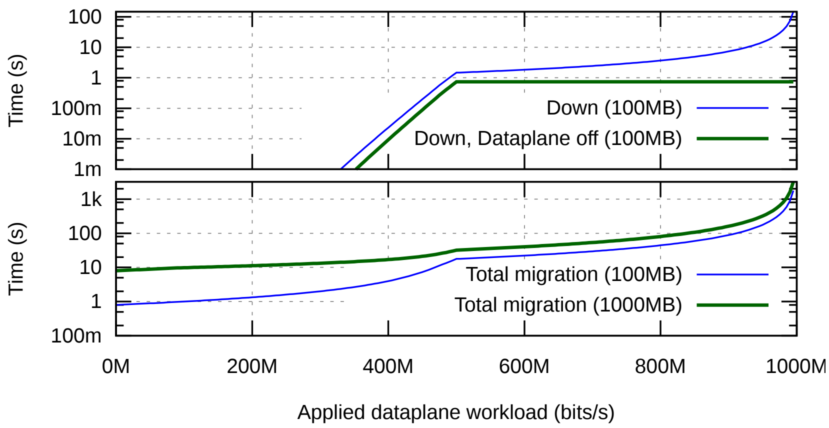


Figure 13: Numerical calculation of total migration times and last-round downtimes, based on an optimistic model of delta-based VM live migration. [127]

the memory delta of round 0. To obtain the time to transfer the memory of round i, we can use Equation 7.

$$t_i = \frac{m_i}{r_c - r_{DP}} \tag{7}$$

During the time t_i, memory will have been invalidated by incoming packets stored in the ring buffer with a traffic volume of $r_{DP} \cdot t_i$. As the buffer eventually wraps around, the invalidated memory will not exceed the size of the buffer ($c_{mbuf} \cdot bufsz$), here, $bufsz$ is the size of a packet in a buffer, and c_{mbuf} are the number of packets which can be stored in the ring buffer. Equation 8 provides the invalidated memory which has to be transferred in the next round.

Delta round sequence

$$m_{i+1} = min\{r_{DP} \cdot t_i, c_{mbuf} \cdot bufsz\} \tag{8}$$

Analysis of a 1Gbit/s link limit

In an example, we assume $r_c = 1Gbit/s$, like for the WAN bottleneck m in Section 4.4.1. The ring buffer is sized with $bufsz = 1400byte$, and $c_{mbuf} = 64K$, like in our implementation in Section 4.5. We let the modeled delta-based migration mechanism do 10 rounds of delta transfer before stopping the VM execution in the last round. Figure 13 shows the resulting total migration time (bottom graph) and the downtime in the last round (top graph), both depending on the applied data plane workload. The total migration time is given for two different snapshot sizes, $100MByte$ and $1000MByte$. Regarding the downtime, we only consider the $100MByte$-sized snapshot, however, we also consider a scenario where the data plane is *inactive* during the last snapshot round, so that the entire link capacity is available for snapshot transfer.

Conclusion

With increasing data plane capacity, the downtime exponentially grows up to $\approx 1s$ at $\approx 500Mbit/s$. The cut at this point is caused by the ring buffer wrapping around. At this point, the downtime slightly grows, as the delta transfer has to compete with data plane traffic, finally reaching infinity. In the scenario with the data plane turned off during downtime in the last round, it remains constant at $1s$.

We conclude that downtime incurred by delta-based state resynchronization when using ring buffers for network I/O cannot be considered as seamless according to our definition in Section 2.7.1.1, as beyond $450 Mbit/s$, the downtime will be larger than $16ms$. In the analysis, we have considered the theoretical minimum of migration time and downtime. It is a *lower bound*, which does not include other state maintained by the VM, as well as several other challenges which are probably further deteriorating the performance, CPU overhead caused by maintaining the shadow page table, transport protocol overhead (e.g. TCP slow start) and latency (e.g. while acknowledging a delta round). These challenges lead to much higher downtimes even in intra-datacenter environments [1].

4.5 IMPLEMENTATION

This section describes an implementation of the SliM mechanism described in Section 4.3 as a *framework* for vNF development, written in the C programming language. The framework has been used for our testbed-based evaluation (Section 4.6 and 4.7), all sources are available on GitHub[7]. We have selected the DPDK [35] (Section 2.8.3) as a platform for packet I/O to achieve high performance, for example by circumventing kernel overhead through a userland network card driver.

Implementation overview

The SliM framework is distributed software, and comprises (1) the VIE running at the vNF instance's computing node, (2) the NFV controller, and (3) the SDN controller running on a bare-metal switch. Instead of running SliM's VIE on the hypervisor, we have decided to implement it in the virtual machine's *guest* context to avoid frequent context switching (Section 2.8.4) between host and guest for SliM operations.

Platform requirements

Regarding the virtualization platform, we chose `libvirt`/`Kernel-based Virtual Machine (KVM)`, which is open-source and well-known in the community, nevertheless, only minor modifications are required when using an alternative virtualization platform, as the guest-based VIE only uses standard network interfaces for guest-host communication. For the required traffic redirections already described in Section 4.3, traditional Layer 2 networks are not sufficient, we require a more precise traffic control. Therefore, we implement an SDN controller using the OF-DPA interface on a commercial-off-the-shelf bare-metal switch. The implementation of the controller on the forwarding device itself, instead of on a remote server using OpenFlow, is beneficial to SliM's traffic redirection sequence during NF migration, for example, the drain packet can be injected much faster after switchover.

Implementation architecture

Figure 14 illustrates the SliM framework's VIE with a layer model. By implementing the VIE in the guest context, the hypervisor can be kept unaware of the SliM process and only needs to accept VM startup and shutdown requests. The framework *requires* the vNF instance's implementation to use DPDK. After loading the environment abstraction layer (EAL) of DPDK [35], the vNF instance can load the SliM framework's VIE code as a shared object (.so) library, and initialize it with the NFV controller's address and callback functions for snapshot and statelet

7 https://github.com/nokia/SliM

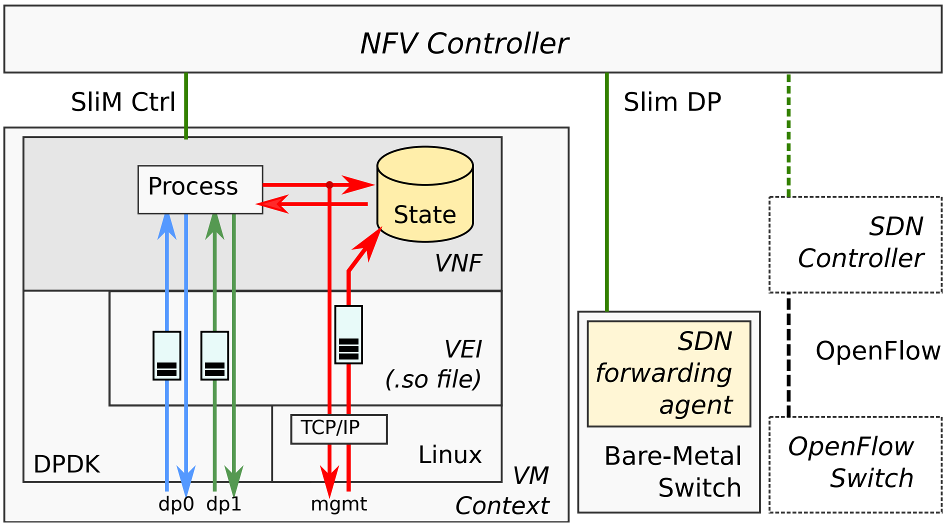


Figure 14: Operating system and software library model of the proof-of-concept implementation. The elements in the dashed lines have been implemented, but have not been used in our testbed. [127] (Adapted)

installation. After the VIE library has been loaded, it cares about all vNF migration tasks in lieu of the hypervisor or the vNF instance, given that the vNF instance's code correctly implements the statelet interface. The SliM control traffic and the statelet and snapshot stream are transferred using TCP over the virtual control network, using the Linux network stack instead of DPDK.

API changes to DPDK

The SliM framework allows the vNF implementation to use the default API of DPDK with two exceptions: First, `lcores` (CPU cores used by DPDK processes) must be requested via SliM and cannot be claimed via DPDK. Furthermore, SliM provides interfaces for packet reception which must be used, as SliM must implement the `pkt_q`.

4.5.1 *Statelet Processing Flow*

The SliM framework makes uses the `rte_ring` stucture in DPDK to implement `st_q_in` and `st_q_out`. The thread-safe ring-buffered queue implementation lets multiple `lcore` processes of the data plane concurrently enqueue statelets to `st_q_out`, by calling the `announce_statelet()` function of the statelet interface. The statelet enqueueing only requires a few CPU operations which should not deteriorate data plane performance. Our implementation uses a single statelet dequeue thread, which, after dequeueing, encodes the statelets on a byte stream using the type-length-value (TLV) scheme: A 2-byte *type* metadata can be announced in our implementation, together with the statelet. To encode a statelet, the type and 2-byte length of a statelet are written to the stream before the contents of the byte vector.

Statelet stream encoding

Statelets in a TCP segment

Simultaneously, the statelet stream is transferred to the destination using maximum-sized TCP segments only (except the last segment when the stream is closed). Our implementation currently uses the Linux TCP stack, although alternatives could be used like the DPDK-based `mtcp` [81]. For this, the stream is buffered at the source until TCP's maximum segment size (MSS) has been reached. Thus, a single TCP segment can carry hundreds of statelets. If l is the average statelet size and M the MSS of the TCP stack (commonly defined by the operating system), the number of statelets which can be carried by a single TCP segment is given in Equation 9.

$$s = \frac{M}{l + 2Byte} \tag{9}$$

For example, if the MSS M is configured with 1400 bytes and the statelets are 12 bytes in average, $s = 100$.

The encapsulation of s statelets in a single TCP segment and IP packet requires the intermediate devices on the path to the destination to only do one forwarding operation per s statelets. Furthermore, it saves overhead caused by header encapsulation and inter-frame gaps on the link. The metric s can be used compare the packet rate of the data plane to the packet rate of the statelet stream. This could be important when comparing SliM to Duplication in scenarios where *packet rates* are considered the bottleneck.

TCP accumulation and latency

The TLV concatenation and buffering of statelets up to the MSS can result in large delays of the statelet stream transmission between the source and destination instance. This would be even aggravated by TCP retransmissions, probably occurring as a flow control measure. However, the path of the statelet stream is not delay-critical *until it is closed*, resulting in an implicit flush of the buffered statelets and the immediate transmission of the remaining segment.

4.5.1.1 *Possible extensions*

Discarding statelets

Several extensions to the statelet approach can be implemented. If statelets are buffered in queues or in the stream at the source, it is possible to delete them before transfer. For example, this could be used to remove statelets which become irrelevant, as their state has already been overwritten by subsequent state operations, decreasing the traffic of the statelet stream. Unfortunately, such a mechanism would require significant additional overhead, for example by maintaining pointers to past statelets or doing an iteration over them. As the expected additional programming effort and CPU overhead would not justify an expected traffic reduction, our implementation does not include it.

GZIP compression

To further decrease the size of the statelet stream, it could be *gzip*ped. Given the corresponding libraries, a `gzip` encapsulation is relatively easy to implement, however, it puts a challenge to the statelet dequeue thread performance, thus it is also not part of our implementation.

4.5.2 *Isolated Snapshots*

Tables can be freezed and thawed

To meet the requirements on the system model in Section 4.3, we have implemented a mechanism to make *isolated snapshots* of the *rte_hash* hashtable implementation of DPDK. From an interface point of view, the hashtable can be *freezed* in a single, atomic operation. After freezing, it is possible to do *freeze-read* operations on the table *from a separate thread*. These freeze-read operations return exactly the state the table had when it was freezed. Other threads, like the data plane, can continue using the read and write operations as usual. They can modify the current state without influencing the freezed one. To make an isolated snapshot, we can freeze the current state and immediately iterate over the list using freeze-read operations only, while data plane threads continue with read and write operations on the table. After taking the snapshot, the table is *thawed* again, which results in freeze-read operations to behave as the default-read operations.

Implementation aspects

Our implementation `slim_freeze_hash` *decorates* DPDK's `rte_hash` implementation. It provides the same interface like `rte_hash`, with additional methods for freezing, thawing, and flags whether to read from the freezed or the default state. Internally, it is backed by `rte_hash`.

Formal structure

In the following, we describe the isolated snapshotting state and procedure model formally. For every key $k \to v$, `slim_freeze_hash` maintains an entry in the format

$$k \to (v_a, g_a, v_b, g_b) \tag{10}$$

in the backing `rte_hash` structure. v_a and v_b are two different instances of the value which are stored. g_a and g_b are positive integer values which we refer to as *generations*. Along with the per-entry generations, we maintain two per-table positive integers h and i, the *table generations*. When initializing a new table, they are both initialized with 0. When the table is freezed, i is incremented by one, when the table is thawed again, h is incremented by one. Thus, i is one ahead of h while the table is freezed, but h catches up on thawing the table. In the following, we describe the behavior of the table operations formally.

Generations

Operations

- If we put an entry $p \to q$ to the table which is not yet present, our implementation creates an entry $p \to (q, i, 0, -1)$ in the backing `rte_hash`, where the "0" denotes a null pointer (a missing entry).
- If we put an entry $a \to b$ to the table for which an entry (v_a, g_a, v_b, g_b) already exists at k, it is replaced in the backing entry with (v_a, g_a, q, i), if $g_a > g_b$, with (v_a, g_a, v_b, g_b) otherwise.
- If we *default-read* an entry for the key a, the entry (v_a, g_a, v_b, g_b) is obtained from the backing table. We then return v_a if $g_a > g_b$, otherwise we return v_b. Thus, we return the entry with the highest generation.
- If we *freeze-read* an entry for the key a, the entry (v_a, g_a, v_b, g_b) is obtained from the backing table. We then return v_a if $g_a > g_b \wedge g_a \leq h$, v_b if $g_b \leq h$, and 0 otherwise. Thus, we return the entry with the highest generation, but not exceeding h. If no such entry is available, we return null (the entry is not present).
- Deletion of a table entry is done by just deleting the entry in the backing table, if the table is are not in the freezed state. If it is in the freezed state, the

aforementioned put operation with a null pointer ("0") is done instead of the deletion.

Repeated use

Using the generation counters, it is possible to implement many subsequent *freeze* and *thaw* operations. This is required after a partial migration, where the VM data plane traffic continues at the source, and must be ready for further freezes. It is also required to be able to resume normal the data plane operation if a migration fails for various reasons, and to retry snapshotting at a later point in time. Furthermore, for the freeze and for every table operation, only a fixed number of additional operations is necessary in the table structures, no iteration over entries etc. is needed.

Iteration and serialization

Iteration over the entries is done by the help of the backing `rte_hash` table. However, it does not guarantee thread safety, as no entries must be deleted by other threads, while iterating over the elements with the current thread (this would break the pointers in the linked list). The `slim_freeze_hash` table is thread-safe during iteration over the freezed state entries, as no entries are deleted in the backed table during the freeze, they are set to the null pointer instead. Added elements during the iteration are not breaking the linked list, but behave indeterministic (some of them are being iterated over, some of them are not). However, these new items have been added during the freeze, and they are ignored, as their generation exceeds h.

Cleanup process

An open point is that for possible entries deleted during a freeze (which have been set to the null pointer instead), a cleanup should be triggered to reduce the number of entries in a hash table. However, it is only required if the state operation continues after a thaw, for example after a failed full migration and a split. Furthermore, the lack of such a mechanism only constitutes a minor penalty regarding memory occupation if only little state has changed during a freeze period. A possible workaround is to place null-pointered elements in a queue, and delete them after the freeze, but this workaround would either not be thread-safe, or must be done by data plane processes.

4.5.3 *SDN Controller Implementation*

The SliM SDN forwarding agent used for the evaluation has been implemented as a Linux application for bare-metal switches. It uses OF-DPA for controlling network forwarding. Instead of using OF-DPA via OpenFlow on a remote controller, we decided to use the native OF-DPA API on the bare-metal switch (similar to the usage in Section 6.1.1). The execution of the agent locally on the switch using an API without a network in between shall provide a better ASIC control for time-critical modifications, for example the drain packet injection described in Section 4.3.3.4 must immediately follow the switchover operation.

As an alternative, we have also implemented a forwarding agent for the *ryu* OpenFlow controller in order to be able to use SliM with OpenFlow switches. The implementation has been successfully tested with Open vSwitch. However, using OpenFlow, we likely have less control over time-critical sequences of operations[8].

4.5.4 *NAT vNF Implementation*

As an example, we have implemented two NFs using the SliM framework and implementing the statelet interface.

First, we have implemented a *masquerading* NAT network function, as described in the examples in Section 4.2. Like in the example in Figure 8, the NAT NF implementation uses 2 interfaces, an *exterior* interface (for example a company's LAN), and an *interior* interface (for example the Internet), both having an IP address configured in the NF at startup.

The NF waits for packets sent from interior hosts to the Internet. It relays the packets to an exterior network while masquerading the internal hosts behind the external IP address. The NF supports TCP and UDP sockets, as well as port translations by incrementing port numbers starting from 1024. The NF supports ARP request/response handling, however, ICMP NAT and IP fragmentation is currently unsupported.

The vNF's state is comprised of a NAT table, an ARP table, and a packet counter for every host in the interior network (this might be used for accounting reasons). A statelet of **18 bytes** is announced on every new socket, containing the interior adress and port and the remote address and port, in order to fill the entry of the NAT table. Furthermore, a statelet of **10 bytes** is announced for every new ARP response, containing the link-layer address and the IP address of the answering host. Finally, for every change of the hosts packet counter, which occurs on every accepted packet, a statelet of **12 bytes** is announced. The counters have also been used in our evaluation to ensure state consistency of SliM.

4.5.5 *Mobile Handover Gateway vNF Implementation*

The second NF we have implemented is a mobile handover gateway. The NF does not use standard protocols used for mobile packet cores, but models the *handover* functionality for example in S-GWs and shall demonstrate the feasibility of using SliM to migrate even NFs with time-critical requirements.

The gateway mediates between mobile cells (first NF interface) and a core IP network (second NF interface). It expects IP traffic from cells to be tagged with a 2-byte cell ID. Furthermore, it is announced in the packet whether the current mobile subscriber is to be handed over to another cell. Traffic from the cells is always relayed to the core network. Traffic from the core network must, however, be tagged with the correct mobile cell, before being sent to the cell-facing interface (first NF interface). Therefore, the NF must keep track of the cell the NF is currently

8 This would e.g. require bundle support for PacketOut messages, however, bundle support appears to be only intended for modifications (`https://mail.openvswitch.org/pipermail/ovs-discuss/2016-February/040058.html`).

associated to in a state table, thus, it must announce a statelet for every handover (50 bytes).

4.5.6 *About Traffic Prioritization*

Problem statement

If we ignore signaling traffic, different packet flows can be distinguished which are in operation during a deterministic-replay-based instance migration, the *data plane workload* (we summarize all data plane interfaces here) and the *state transfer* flow. The state transfer flow can furthermore be distinguished into the *snapshot stream*, and the *resynchronization stream* (the statelet or dupe-packet stream). An open question is if a prioritization of any of the flows can improve the performance of the overall migration mechanism.

4.5.6.1 *Prioritization between data plane workload and state flow*

Avoiding TCP retransmissions

A prioritization of the data plane workload over the state transfer traffic can be applied, it might slightly reduce delay and jitter on the data plane. However, it should be avoided with any NF workloads where congestion may occur. Upon congestion, it will lead to the snapshot and resynchronization streams to drop packets. If TCP is used, like in our implementation, this will cause retransmissions. Retransmissions, however, aggravate the congestion, and probably result in infinite duration and eventual migration failure, although the data plane will not drop packets or cause any jitter.

Avoiding data plane drops

A prioritization of the state transfer streams over the data plane should be avoided as well, in order to ensure seamless behavior. The prioritization will result in a fast migration process, but cause a downtime, as the state transfer traffic will quickly supplant the data plane, leading to the drop of nearly the entire workload during migration.

We conclude that prioritization cannot mitigate the congestion problem here, it will lead to either migration failure or loss of seamless behavior. Thus, we do not apply prioritization between data plane workload and state transfer streams in our implementation.

4.5.6.2 *Prioritization between snapshot stream ans resynchronization stream*

Prioritization does not harm state migration process

Finally, priorities might be applied between the snapshot stream and the resynchronization (resync) stream. When the snapshot stream is prioritized, nearly all statelets/dupe-packets must be buffered in `st_q_out` of `srcInst`, as the resync stream does not have any transfer capacity available until the snapshot stream has been closed. The reason is that the snapshot stream is *greedy*, the TCP stream will claim as much bandwidth as possible until the entire snapshot has been transferred, and will supplant the resync stream. In exchange, the resync stream can use the entire available bandwidth *after* snapshot transfer to keep up. When prioritizing the resync stream, statelets/dupe-packets are quickly transferred, and will wait at `st_q_in` at `dstInst`. Prioritizing the resync stream will not lead to an expulsion of the snapshot stream, as it only claims enough bandwidth to cover the current

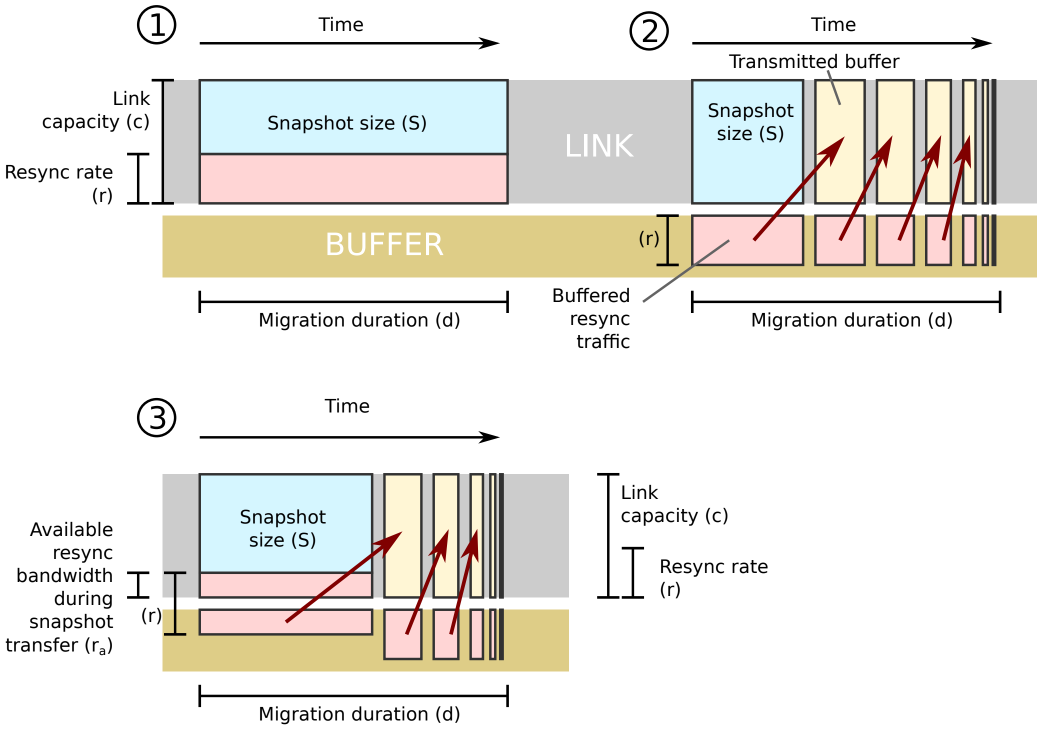


Figure 15: Illustration of sizing for different prioritization strategies between statelet and snapshot stream.

rate of resync data. Thus, any prioritization will not harm the success of a seamless migration. However, the question remains whether migration duration can benefit from a prioritization.

Prioritizing the resync stream

Figure 15 illustrates the transfer of the two state streams over a link (gray area) with a total limited capacity for state transfer[9] c. Parts of the state which cannot be transferred yet can be buffered at `srcInst` in `st_q_out` for later transfer (brown area). We model the time in the horizontal dimension, and the bandwidth used in the vertical dimension. We furthermore assume that the resync rate r is constant during migration[10].

In the first scenario 1, the resync stream is prioritized. The stream sends with a rate r, it cannot send with a higher rate, thus it is not greedy and will not supplant the snapshot stream, which can use the remaining rate $C - r$ to transfer. In this scenario, the migration duration is given by Equation 11.

$$d_1 = \frac{S}{c - r} \tag{11}$$

Prioritizing the snapshot stream

9 In this section, c is the remaining link capacity if we subtract the bandwidth required for data plane workload.

10 This is likely not true with real workloads, whether we use dupe packets or statelets for resync. However, if we assume a high-performance NF instance processing a large number of sessions, the rate should be approximately constant.

In the second scenario ②, the snapshot stream is prioritized. As it is greedy, it will consume the entire link capacity, supplanting resync data, which must be buffered at `st_q_out` at `srcInst`. After the snapshot has been transferred, the buffered contents of the resync data can be transmitted with the full rate of c. However, during the transfer of buffered resync data, newly arriving resync data must be buffered and transferred thereafter, and this step must be repeated until no more buffered resync data is left, in order to complete the migration. The migration duration d_2 is given by Equation 12 and equals d_1 (the extended derivation is given in Appendix A.1).

$$d_2 = \sum_{i=0}^{\infty} \frac{S}{c} \cdot \left(\frac{r}{c}\right)^i = \frac{S}{c-r} \tag{12}$$

Scheduling

The third scenario ③ is a hybrid prioritization scheme, where the snapshot is transferred with a rate $c - r_a$, so that the statelet stream can be partially transferred with a rate r_a, but must be partially buffered with a rate of $r - r_a$. In this scenario, the migration duration d_3 is given by Equation 13 and equals d_1 and d_2 (the extended derivation is given in Appendix A.1).

$$d_3 = \frac{S}{c-r_a} + \sum_{i=0}^{\infty} \frac{S}{c-r_a} \cdot \frac{r-r_a}{c} \cdot \left(\frac{r}{c}\right)^i = \frac{S}{c-r} \tag{13}$$

According to Equation 13, the rate r_a does not have an influence on the overall migration duration, thus, no specific scheduling configuration can improve it. Furthermore, $d_1 = d_2 = d_3$, thus, we conclude that also strict prioritization does not have any effect according to our model.[11]

In our implementation, we therefore do not use prioritization between statelet and snapshot stream, they are transferred simultaneously over two TCP streams using best-effort transfer. Using two simultaneous streams has a positive effect on the migration duration, as both streams quickly reach the maximum or the required TCP transfer rate when using TCP slow start. However, using too many streams is harmful to TCP fairness [101], probably negatively impacting other processes in the network.

4.6 EVALUATION SETUP

In this section, we describe a testbed evaluation setup based on the implementation in Section 4.5. The evaluation is conducted in a testbed on state-of-the-art NFV hardware. The goal is to obtain more reliable results on the performance of SliM, by taking into account properties of the operating system, the hardware, and transport protocols, which have not been taken into account in our analysis in Section 4.4.

Duplication mode

To compare the SliM framework with duplication-based state resynchronization, we have implemented a duplication-based state resynchronization *mode* for the

11 In the model used, we ignore any transmission overhead, like TCP retransmissions acknowledgements, and TCP slow start.

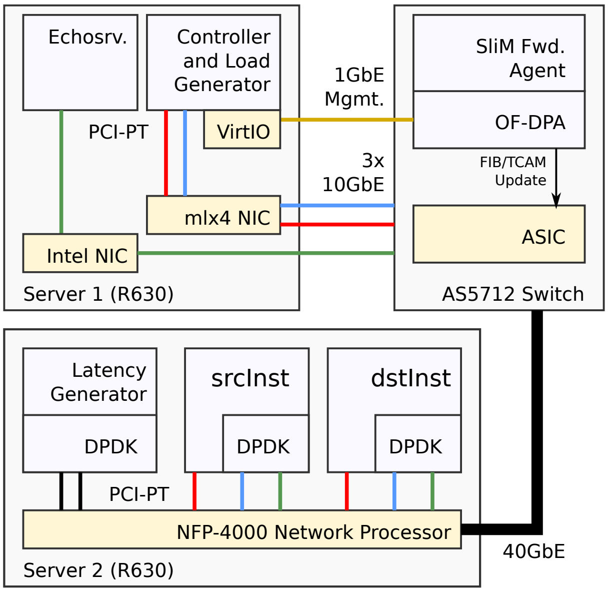


Figure 16: Physical model of the evaluation setup [127]

framework. If configured for duplication-based mode, the framework does not request statelets, but duplicates ingress packets and sends them over to the destination NF for reprocessing. Although frameworks for duplication-based state migration are already available [56, 57], we decided no to use them to be able to do a comparison between the *principles*, and not deteriorate the accuracy of our performance results by implementation details. Based on the findings in the analysis about a lower bound of VM live migration performance under data plane workload (Section 4.4.2), and the non-seamless results of related work even in high-throughput and low-latency scenarios [1, 26, 154], we do not consider delta-based VM migration in our testbed evaluation.

4.6.1 *Hardware Setup*

Physical testbed setup

Figure 16 illustrates our hardware and VM setup used. Our testbed comprises two *Dell PowerEdge R630* dual-socket servers and an *Edge-Core AS5712-54x* bare-metal switch. The two CPU sockets of the server each hold an *Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz* processor with 10 physical[12] cores each. The servers run an *Ubuntu 16.04* operating system, with KVM/QEmu and `libvirt`.

Server 1

Server 1 is used for workload generation and measurement. It is equipped with a *Mellanox ConnectX-3 (mlx4)* and an *Intel 82599ES* NIC. Each card has two 10 Gigabit Ethernet ports, however, from the Intel card, only one port is used. Server 1 carries two VMs, a *controller and load generator* VM, and an *echoserver*. The controller and load generator VM operates SliM's NFV controller and a load generator used to create data plane workload for the virtual machines. We decided to place the controller and the load generator on the same machine in order to

12 non-hyper-threading

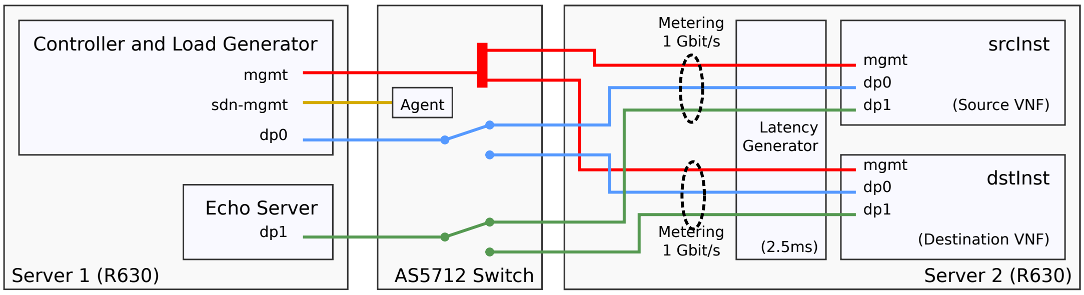


Figure 17: Layer 2 flow model of the evaluation setup [127]

exploit advantages in the automation of the tests. The echoserver is also part of the workload generator, however, it reactively answers to received packets with other packets, and does not send or receive commands from the automated evaluation framework. Both the echoserver and the controller/load-generator use the Linux kernel network stack instead of DPDK in order to demonstrate the compatibility of the vNF implementations and the SliM framework with the stack. The VMs are directly accessing the NICs via PCI-PT which avoids virtualization I/O overhead (Section 2.8.2). The controller/load generator VM claims the two Mellanox ports, the echoserver uses one of the Intel interfaces for data plane I/O.

Server 2 Server 2 is equipped with a *Netronome Agilio CX* NIC with a 40 Gigabit Ethernet network interface, which carries an `NFP-4000` network processor. The NIC can spawn multiple PCI *virtual functions (VFs)* (Section 2.8.2) for the VMs, and flexibly do packet forwarding based on its programming. Server 2 carries the `srcInst` and `dstInst` VMs and a latency generator. Although `srcInst` and `dstInst` run on the same node in our testbed, the communication between them will be configured like if they are in remote datacenters by using a latency generator. It furthermore carries the `Sloth` latency generator described in Section 4.6.5. All VMs claim VFs from the Agilio LX card via PCI-PT. `srcInst` and `dstInst` each claim three interfaces: two data plane interfaces using DPDK and SliM, and one interface using the Linux kernel for management. The latency generator claims two VFs for its purpose.

Switch The bare-metal switch runs *Open Network Linux* (ONL) with SliM's OF-DPA-based SDN forwarding agent described in Section 4.5.3. The management link of the switch is connected to the controller/load-generator VM via VirtIO (yellow link), so that the SliM NFV controller is able to send redirect commands to the switch. The 10 Gigabit Ethernet ports of the NICs passed through to the VMs are connected to the switch, like the 40 Gigabit Ethernet link of the Netronome card.

4.6.2 *Flow Model*

L2 flow model Figure 17 illustrates the network flow model we have configured on the hardware setup described in Figure 16. The management domain (red line) is a switched Layer 2 network between the controller VM, `srcInst` and `dstInst`. It is used for signalling and for state transfer traffic. `dp0` (blue line) is the first data plane interface of the vNF. By default, it directly forwards traffic from the controller/load-generator VM to srcInst and back. `dp1` (green line) is the second data plane interface respectively, which forwards traffic from the *echoserver* to `srcInst` by default. By request of the SliM framework, the traffic of `dp0` and `dp1` can be redirected to the interface of `dstInst` according to the SliM redirect sequence described in Section 4.3.3 and back.

WAN scenario modeling

Like assumed in the analysis, our testbed is supposed to resemble a WAN scenario, in which a bandwidth-restricted, high-latency link is shared between the data plane and the state migration traffic. Therefore, the common bandwidth of the three links of an instance is limited to 1 gigabit per second (Figure 17, dashed rings) with a burst size of 20ms. This is done with a *meter* in the NFP-4000 flow processor, which drops all packets beyond this limit. Additionally, the latency generator (Section 4.6.5) adds a latency of $2,5ms$ in each direction between `srcInst` and `dstInst`, which will result in a round-trip time (RTT) of $10ms$ between the load generator and the echoserver, or for management traffic between `srcInst` and `dstInst` It will also result in an RTT of $5ms$ for any traffic between the controller and the instances, or the switch and the instances. We have selected the configured delay of $10ms$ based on findings about the speed of typical intra-European datacenter links[13].

4.6.3 *Evaluation Runs*

Based on the described hardware setup (Figure 16) and flow model (Figure 17), we conduct automated evaluation runs to obtain performance data from the SliM framework and its duplication mode. An evaluation run begins at time s with a booted-up `srcInst`. The load generator now starts to apply workload by sending a stream of *time-stamped* packets with a specific rate to `srcInst`. The workload is processed by `srcInst`, sent to the echoserver, answered by the echoserver, processed by *srcInst* again, and sent back to the load generator. The load generator finally receives the packets to determine performance data. `dstInst` is also booted up, however, the VM is still idle and has no state. While the workload is still applied, we trigger a SliM instance migration to `dstInst` at $s + 5sec$. The experiment continues with workload being applied until $s + 15sec$. Before taking down the experiment for the next round, additional statistics are gathered from the vNF's log output by the controller. For different setups (described in Section 4.7) and for different data plane throughput, we conduct an experiment using an automated script. To obtain results with a high confidence, an experiment of any specific parameter combination is repeated 13 times.

4.6.4 *Evaluation Workload*

Workload description

In our setup, the workload generator creates and receives UDP/IP packets with a specified packet size and at a given rate, so that the packet rate times the packet size results in the desired throughput in bits per second. Upon packet egress, a

13 http://www.verizonenterprise.com/about/network/latency/

sequence number and the current system's timestamp is written into the packet. The sequence number allows tracking lost packets in the packet stream, while the timestamp – in combination with the sequence number – can be used to determine the RTT of every packet.

Loadgen behavior

For the NAT NF implementation, the workload generator resembles a group of IP hosts in the interior network, opening 128 sockets to 4 exterior hosts simultaneously (e.g. VoIP servers in the Internet), where the latter are resembled by the echo server. The NAT NF translates the sockets and forwards the packets to the echo server, the echo server answers with UDP packets to the NAT NF. Finally, the NAT NF translates the socket of the received packets back, before forwarding it to the load generator.

Special behavior for the handover NF

To evaluate the mobile handover NF, the load generator resembles 256 mobile subscribers, which move very fast between cells, with a mean handover frequency of 2 seconds. Furthermore, a packet is considered as lost if it has not been sent to the cell the subscriber is currently associated to, or has been associated to during the last $7,5ms$. With the special handover NF load generator behavior, we intend to demonstrate the capability of SliM to quickly make state changes caused by certain packets available to returning packets. The grace time of $7,5ms$ shall account for the RTT between the controller and each NF instance, which constitutes a lower performance limit.

Packet sizes, snapshot size

In our evaluation, the load generator either creates packets with $1400Byte$ or $512Byte$, depending on the scenario. A packet size of $1400Byte$ is almost the maximum transmission unit (MTU) in the Internet ($1500Byte$), thus it is rather an optimal case. Therefore, we also consider a packet size of $512bytes$, which is approximately the average packet size in the Internet[14]. To demonstrate that SliM also migrates VMs with larger states, we have added $20MByte$ of filling zeroes on top of the serialized snapshot. If we assume a session state of $50Byte$, this corresponds to hundreds of thousands of simultaneous sessions, which we could not emulate with our load generator for performance reasons.

4.6.5 *Sloth: Latency Generator*

The communication between nodes in centralized testbeds are commonly subject to sub-millisecond delays. However, to evaluate the SliM implementation under WAN-like conditions in such a testbed, packets on the modeled WAN link must be delayed, which can be achieved with a *latency generator*.

In a first attempt, we evaluated the widely-used `NetEm` network emulator [65] as a latency generator However, in an initial test, it was not possible to achieve TCP transmission rates beyond 300 Mbit/s with a latency of $10ms$ with `NetEm` on our lab hardware. Furthermore, no latency generator has been available making use of DPDK for accelerated packet I/O.

Sloth is a DPDK-based latency generator, which operates similar to *testpmd* in the sense that packets are forwarded between interface pairs. Upon every entering

14 https://www.caida.org/research/traffic-analysis/pkt_size_distribution/graphs.xml

packet, it waits a fixed latency provided at startup before the packet is transmitted on the other interface. Sloth is not capable to add random latency or jitter, however, a constant latency is sufficient to model the physical transmission limits of a WAN scenario which we envision to do with the software.

Sloth operates by placing the DPDK `mbuf` structures of incoming packets in a large ring buffer with a timestamp (producer operation). Concurrently, packets with a timestamp older than the current time plus the latency are transmitted on the paired interface (consumer operation). Sloth can be configured in the 1-core mode and the 2-core mode. In the 1-core mode, every CPU core concurrently executes the producer and consumer operation of one interface, while in 2-core mode the operations are separated on 2 cores for each interface.

Sloth is available on GitHub[15] since February 2017. Since our initial commit, the latency emulator *DEMU* has been published with a similar functionality [2].

4.7 EVALUATION RESULTS

In this section, we present the performance evaluation results of the SliM framework by doing a performance measurement as described in Section 4.6 of two NFs. Regarding the first NF, we also compare the performance of the SliM framework to the performance of packet-duplication-based deterministic replay, by enabling a *duplication* mode in the SliM framework and comparing the results.

4.7.1 *Metrics Used*

We assume that it is crucial for a good performance of the mechanism that the NF operates *seamlessly* according to our definition in Section 2.7.1.1, even during migration of the NF instance over a bandwidth-restricted and high-delay link. An indicator for seamless behavior is the absence of packet loss or jitter, which both can lead to perceivable disruption of a service. Therefore, we focus on the metrics *latency* and *packet loss* in our evaluation. As both packet loss and jitter (the deviation from the baseline RTT of $10ms$) do not occur continuously, but only for a limited time during state migration, we consider time-dependent metrics. Therefore, we measure packet loss in *seconds of packet loss*, which is obtained by dividing the packets lost during an evaluation run by the current packet rate sent by the load generator (in packets per second). For example, if for one second during migration all packets have been lost, but all other packets have arrived, we have 1 second of packet loss, like if we would have lost half of the packets for 2 seconds. Another method we use is the peak of the average RTT over any $500ms$ interval during the experiment. The metric is obtained by creating averages over all $500ms$ intervals during the evaluation run, and taking the maximum as a result. The latter metric shall illustrate how the average jitter behaves during the "hot phase" in the migration process. Besides latency and packet loss, we also consider *total migration time*, *state traffic volume*, and the maximum size of `pkt_q`.

Seconds of packet loss

Other metrics

15 https://github.com/lnobach/sloth-latency-gen

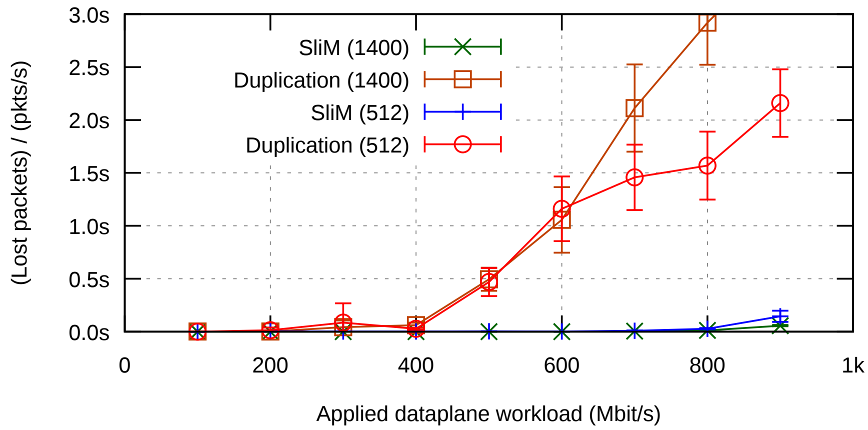


Figure 18: Time of packet loss of the NAT NF when using SliM and Duplication, for different data plane workload, using 1400-byte and 512-byte packets. [127]

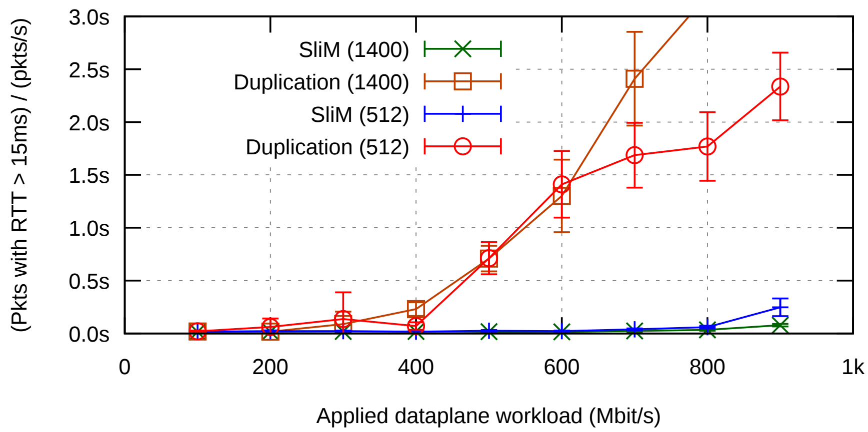


Figure 19: Time of $< 15ms$ SLA violation of the NAT NF when using SliM or Duplication, for different data plane workload, using 1400-byte and 512-byte packets. [127]

4.7.2 *Latency and Packet Loss*

Seamless up to $800Mbit/s$

Figure 18 shows the seconds of packet loss for different workloads for the NAT NF. Additionally, Figure 19 depicts the seconds of packet loss with RTTs not smaller than 15ms for the NAT NF. Up to $400Mbit/s$, we can observe no significant loss for both SliM and Duplication. Starting at $500Mbit/s$, Duplication suffers from serious downtimes. However, SliM can maintain the migration with very small downtimes up to $800Mbit/s$. Here, SliM starts to show downtimes in the area of a few tens of milliseconds, where Duplication already requires $1,5s$ (512-byte packets) and $3s$ (1400-byte packets) of downtime. In Figure 20, the SliM-based handover NF shows that during a period of 100-200 milliseconds, packets are labeled with a cell association which is not more valid for more than $7,5ms$. The staleness is still rather small compared to the packet loss and the $< 15ms$ service level agreement (SLA)

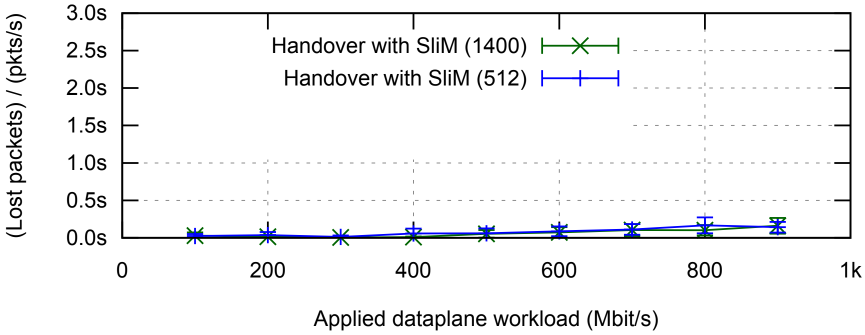


Figure 20: Time of packets received with a cell association older than $7,5ms$ (including lost packets) of the handover NF when using SliM, for different data plane workload, using 1400-byte and 512-byte packets. The range of the Y axis is normalized to $3.0s$ to correspond to Figure 18. [127]

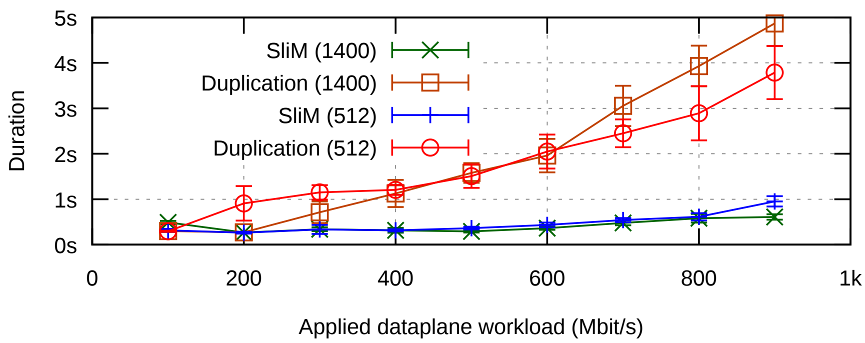


Figure 21: Total migration duration of the NAT NF when using SliM and Duplication, for different data plane workload, using 1400-byte and 512-byte packets. [127]

violation of Duplication with the NAT NF (Figure 19), which would be a lower bound of the Duplication performance here.

4.7.3 *Total Migration Duration*

In Figure 21 and 22 we can observe a significantly higher performance of SliM compared to Duplication regarding the total migration duration. Starting at $100Mbit/s$, the duration is not significantly different. However, starting at $200Mbit/s$, Duplication with a packet size of $512bytes$ requires up to 1 second of time to finish, where SliM can finish the migration in under 0.5 seconds. When increasing the bandwidth to $800Mbit/s$, the migration duration for Duplication increases up to 4 seconds, where SliM can still maintain a migration duration around 0.5 seconds. Regarding the migration duration for the handover NF, only insignificant influence of the applied data plane workload on the migration duration can be observed. This is caused by the statelet rate to remain constant, as a statelet is only announced on every cell transition. Cell transitions, however, are not varied during the experiment, and do not depend on the current packet rate.

More than 3 times faster

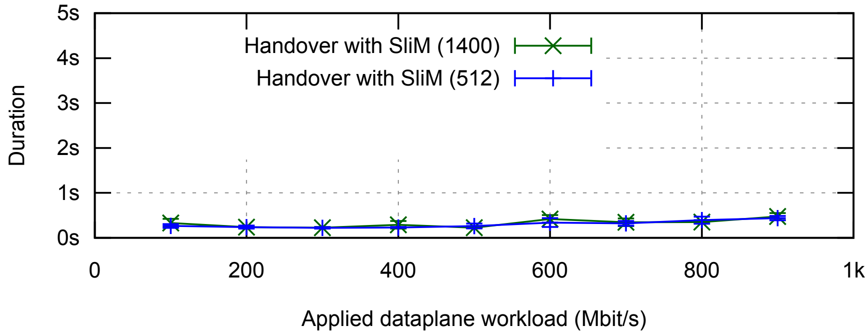


Figure 22: Total migration duration of the handover NF when using SliM, for different data plane workload, using 1400-byte and 512-byte packets. The range of the Y axis is normalized to $5s$ to correspond to Figure 21. [127]

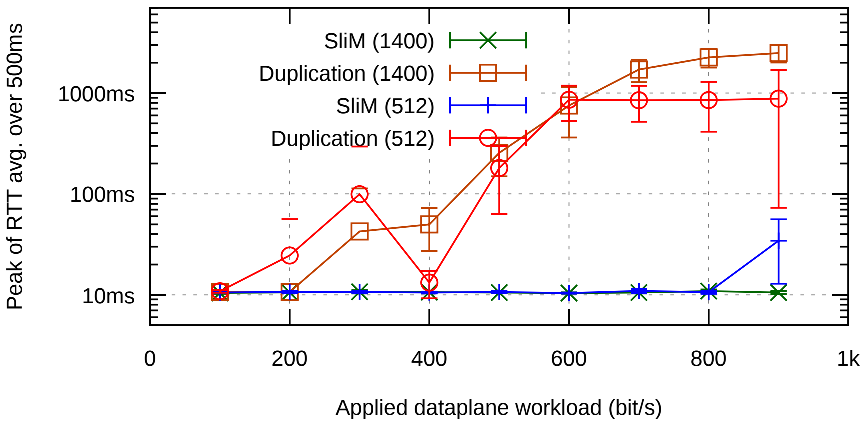


Figure 23: The largest RTT average over any $500ms$ interval during an experiment with the NAT NF when using SliM and Duplication, for different data plane workload, using 1400-byte and 512-byte packets. [127]

4.7.4 *Round-trip time and jitter*

Minor jitter up to $800 Mbit/s$

In Figure 23, we can observe that the baseline RTT of $10ms$, which is added by the latency generator, is highly increased during state migration when using Duplication. Even at low rates, Duplication can cause a jitter up to 100 milliseconds. Wen increasing the rates beyond 500 milliseconds, Duplication adds around 1 second of jitter (512 Byte) or more than 2 seconds of jitter (1400 Byte) to packets processed, making the method unusable for many real-time applications. However, SliM can avoid any significant jitter up to $800 Mbit/s$, only at $900 Mbit/s$ it starts delaying packets.

No congestion even under high link utilization

The cause for the aforementioned increased jitter are packets remaining in `pkt_q`, waiting for their in-order delivery, which starts after the dupe-packet or statelet stream has finished with transmitting its last element. Figure 24 shows the maximum number of packets waiting in `pkt_q`, which has been limited to $64K$ packets in our implementation, dropping packets beyond its capacity. For Duplication with a data

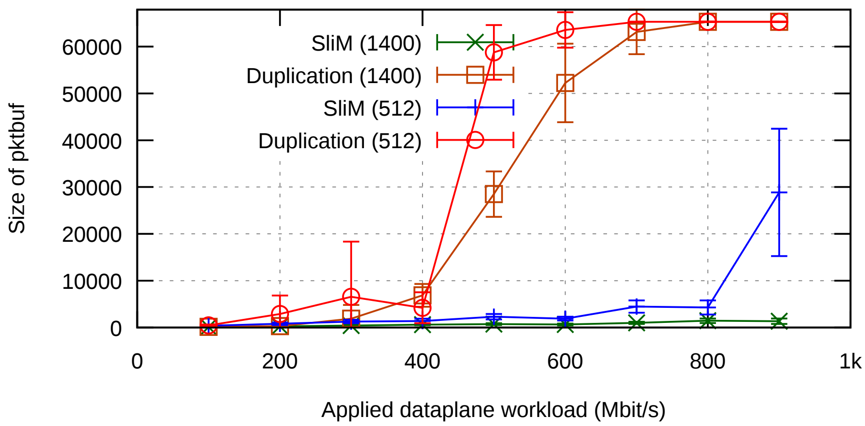


Figure 24: Number of elements in `pkt_q` when migrating the NAT NF using SliM and Duplication, for different data plane workload, using 1400-byte and 512-byte packets. [127]

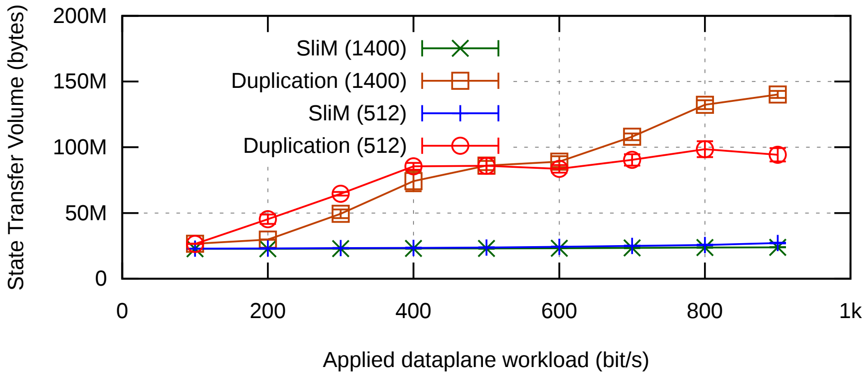


Figure 25: Total state traffic volume when migrating the NAT NF using SliM and Duplication, for different data plane workload, using 1400-byte and 512-byte packets. [127]

plane workload larger than $400 Mbit/s$, we observe that the packet count in `pkt_q` reaches the queue's capacity, which likely also results in packet drops because of overflow. The `pkt_q` is supposed to hold back packets during the drain packet injection and flush of the dupe-packet or statelet stream only, which is supposed to be a very short time only. However, if the stream of duplicated packets is congested at the source, because the required link capacity to transfer them in time is not available, it will finish too late, so that `pkt_q` causes jitter and even packet loss by overflow.

4.7.5 *State Traffic*

Low resync traffic volume

Figure 25 shows the *traffic volume* required for state transfer of the NAT NF, for both SliM and Duplication. The lower bound is the snapshot size, which requires about

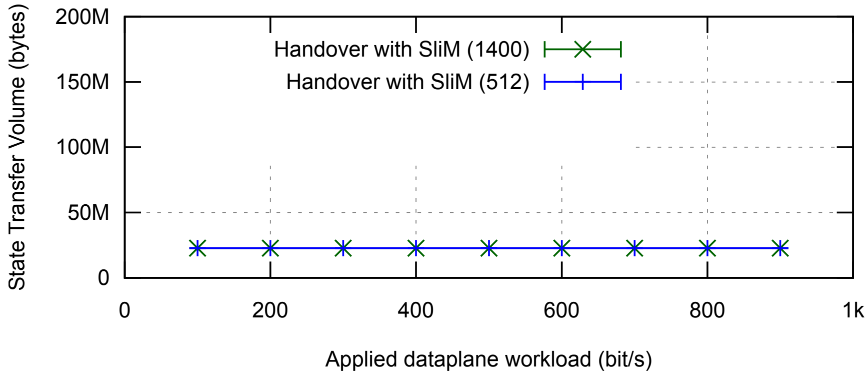


Figure 26: Total state traffic volume when migrating the handover NF using SliM, for different data plane workload, using 1400-byte and 512-byte packets. [127]

$23 MByte$. At $50 MBit/s$, the volume required for state *resynchronization* – duplicated packets – yet makes only a small fraction of the total state transfer volume. However, if we increase the data plane workload, the resynchronization traffic causes the total volume to grow up to $90 MByte$ ($500 MBit/s$). A further increase to $800 MBit/s$ even results in a volume of up to $130 MByte$ when using a packet size of $1400 Byte$. SliM requires adds only a very small fraction of the snapshot size with statelet-based resynchronization. Even in the experiments beyond $900 MBit/s$, SliM maintains a total volume of not more than $30 MByte$.

Figure 26 shows the state transfer traffic volume for the handover NF. As the handover NF's statelet rate is not traffic-rate- but handover-rate-dependent, the average traffic volume remains constant.

4.7.6 *Evaluation Summary*

The evaluation results show a clear performance advantage of the SliM framework in bandwidth-restricted and high-delay situations modeled by the testbed. Duplication already shows first signs of performance degradation at a data plane workload of $300 Mbit/s$ (30% of the available link capacity). At $500 Mbit/s$ and beyond, the degraded performance becomes unacceptable for real-time services. SliM, however, can maintain seamless state migration up to $800 Mbit/s$ (80% of the available link capacity), and only incurs a small jitter at $900 Mbit/s$. The evaluation confirms that SliM can almost triple the data plane workload which the NF can process during migration.

5 ELASTIC HARDWARE ACCELERATION - ARCHITECTURE AND BENEFITS

The ideas and findings presented in this section have been previously published by the author of this thesis [118, 122]. Several sections contain citations from the author's publication [122] which are not explicitly marked. For more information, refer to Appendix B. Cited figures are explicitly marked.

Virtualized network functions (vNFs) sometimes require solving a few complex tasks, for example checking a permission or accessing a database. These tasks are rather infrequent, and only concern a small fraction of the packets in an NF workload, thus these tasks can be considered to be not performance-critical.

Use case complexity vs. performance

However, hardware acceleration has limits in the use case complexity, some complex use cases cannot be implemented efficiently. Pongrácz et al. [133], for example, show how the performance of programmable data planes decreases with the complexity of the use case which is implemented on them. Such a limit must also be assumed regarding FPGAs, as the number of programming gates on a chip is limited, restricting the number of rare or exceptional cases which can be foreseen in the gate logic. This problem is commonly addressed in practice by using a hardware accelerator *directly attached to* a CPU (for example via the PCI interface), offloading too complex tasks to the CPU [55]. This has similarities with OpenFlow in the network forwarding domain [108], which allows sending packets the data plane cannot process to a *controller* in order to execute more complex tasks.

Elasticity vs. heterogeneous processing hardware

However, elasticity [66, 67], an important feature of cloud computing, commonly assumes the computing resources are *homogeneous*: one resource type – commonly the CPU – can efficiently fulfill every computing use case. The tight coupling between commodity CPU and hardware accelerators – *heterogeneous resources* – leads to an elasticity problem, if the *ratio* of simple- and complex-use-case processing resource demand to each other is not previously known and might change during an NF lifecycle. This is because the dimensioning of a computing node regarding CPU and accelerator resources is *fixed* in a datacenter. For example, it is not possible to acquire more accelerator resources per commodity-CPU vNF instance than the ones which are attached to the hypervisor's CPU. A possible scale-out might solve the problem, however, the CPU resources would be under-utilized then. Likewise, a vNF instance might decide to avoid any accelerator resources, for example because of lower performance requirements. This will lead to idle accelerators connected to a highly utilized commodity CPU, which cannot be used by other NFs on the node because of the present CPU utilization. A solution is to decouple hardware acceleration resources from the CPU and *pool* them in the datacenter, making the resources separately available over the network, similar to block device resources in storage area networks (SANs) [59]. The procedure of remotely accessing virtualized FPGA resources has been demonstrated in a proof of concept in previous work [20].

Contribution

In the following sections, we present our proposal of a split-architecture in which the NF processing is split into simple- and complex-use-case traffic via in-network processing (e.g. through a custom P4 pipeline [11]), and handled appropriately by either a commodity CPU or acceleration hardware. Resources for handling the simple- and complex-use-case workloads are *pooled* in a DC independently of each other. In Section 5.3, we analyze the benefits of elastic hardware accelerator provisioning *in general*, these benefits would nevertheless also apply to the suggested architecture.

5.1 ANALYSIS

In the following, we distinguish three processing entities in an NFV infrastructure.

Heterogeneous processing entities

- SDN *in-network processing* refers to network operations being executed by the forwarding devices (e.g. switches) in the SDN infrastructure. When using OpenFlow, these capabilities are restricted by available OpenFlow actions and matchable fields, and often by additional hardware restrictions of the forwarding devices. However, conceptually, we do not need to restrict the SDN capabilites to the ones of OpenFlow, as with pipeline description protocols like P4 [11] – and its upcoming hardware support – it is also possible to split traffic based on customly defined headers beyond the transport layer, or to solve header processing tasks entirely in the network.

- Remaining simple but network-intensive workloads which cannot be processed in-network are forwarded to **HWA** instances, like FPGAs or network processors. Typical workloads for HWA instances include *stateful* operations, *buffering*, *caching*, *traffic shaping*, *encryption*, or *multimedia transcoding*.

- Complex, compute-intensive workloads which are only processed infrequently, and which are less performance-critical, are served by the **commodity CPU**.

In the next section, we provide an overview over several popular NFs as an example how NFs can be mapped to the aforementioned split-architecture concept (see Section 2.4 for further examples of NFs).

5.1.1 *Split Architectures for Important Network Functions*

PPPoE access concentrator

A **PPPoE access concentrator** [104] is important for Digital Subscriber Line (DSL) access networks in Europe. The PPP is designed for communication between two hosts for the purpose of exchanging Network Layer (L3) traffic, and has its origin in dial-up networking. While the customer's DSL router terminates one side, the PPPoE Access Concentrator terminates the provider's side. It listens for PPPoE discovery requests, handles PPP authentication, billing and logging. After successful authentication, the PPP *session* is handled, which has a rather small use case complexity, but is performance-critical, as it must encapsulate/decapsulate high-bandwidth payload data of DSL subscribers. As the PPPoE discovery and authentication steps have a rather large use case complexity but smaller performance requirements, they are rather not appropriate for implementation on HWA and should be run on a commodity CPU. The session stage may be offloaded to HWA, header processing might be also provided directly by in-network processing.

IPSec Endpoint

An **IPSec Endpoint** is used especially in VPNs. Nevertheless, it is comparable to the previous use case: Authentication and key exchange (Internet Key Exchange, IKE) is rather complex and should be solved by a commodity CPU. Payload processing however, i.e. adding/checking authentication headers or encrypting packets in Encapsulating Security Payloads (ESP), can be offloaded to hardware.

Session Border Controller

A session border controller (SBC) [63] is an important use case for HWA, as the performance of dedicated hardware could not be reached by NFV in previous work [112]. The NF acts as an Application Layer Gateway (ALG) for VoIP. For instance, it allows for accessing VoIP devices that are hidden behind a NAT gateway. It often has built-in security features like authentication and DDoS protection. Moreover, it provides audio transcoding of Real-time Transport Protocol (RTP) packets, if necessary. For instance, audio transcoding may be offloaded to HWA. Furthermore, traffic shaping and prioritization could be offloaded if these features cannot be implemented in the SDN.

5.1.2 *Suitable Acceleration Hardware*

ASIC programmability

The performance gain of HWA is generally obtained by leveraging features like parallelization and pipelining which commodity CPUs commonly do not offer in the required level. Elastic provisioning of hardware acceleration for different purposes requires *programmability*. However, ASICs (Section 2.9) are made for a specific purpose and their programmability is very limited, if possible at all. Although the implementation of a carrier-grade use case on *switching ASICs* investigated in Section 6 has been found as feasible, we have discovered limits in functionality. Therefore, we conclude that the use case spectrum of ASICs is limited.

Flexibility of FPGAs

For small quantities, the costs and time-to-market for an NF to become silicon hardware are too high. Therefore, FPGAs are often used as an alternative to ASICs. Like with ASICs, the logic array can be optimized to the specific problem of the NF, allowing for custom processing beyond the performance capabilities of procedural programming for commodity CPUs. The main difference to ASICs is that FPGAs are (re)programmable: The behavior is written in a HDL, for instance VHDL or Verilog, from which a gate list is generated. The gate list can then be installed on the FPGA in form of a *bitfile*. Features like Partial Reconfiguration [169] allow for sharing of the resources of a single FPGA unit among multiple NFs.

Network processors

Network processing units (NPUs) are very heterogeneous, but have several features in common. They are more like commodity CPUs regarding their programmability, but have a larger number of cores with stronger network I/O capabilities than them. The higher performance is achieved by leveraging *parallelization* and *pipelining* between one or multiple cores dedicated to a specific network function. Important vendors include Netronome, Cavium and Mellanox.

Other hardware accelerators

The list of possible acceleration hardware is not limited to FPGAs and NPUs. Depending on the use case, the floating-point single-instruction-multiple-data (SIMD) acceleration of graphics processors may be exploited e.g. for line-rate video transcoding, or network-attached digital signal processors (DSPs) may speed up voice and audio tasks.

5.1.3 *Elastic Workload Redistribution*

Limited virtualization on an FPGAs

While HWA components like FPGAs or NPUs may improve performance of simple use cases, their per-unit costs may be quite large compared to the ones of commodity CPUs (Section 5.4.2). Additionally, elasticity is limited, for example partial reconfiguration on FPGAs requires a minimum set of gates available to implement a network function in a gate region, therefore commonly providing a fixed amount of capacity for the network function independently of the current workload. Thus, it is likely more cost-efficient to run the network function on a (virtualized) commodity CPU, if an HWA resource would be very under-utilized by the current workload. In this case, it is desired to run the workload of the NF completely in a commodity-CPU-based virtual machine, whether complex or simple (Figure 27a). The corresponding network flow is completely directed to the commodity CPU.

Workload redistribution

If performance requirements increase to a level where a HWA resource can handle the simple-use-case workload more efficiently than a commodity CPU due to higher utilization, this workload is distributed to such a resource (Figure 27b). The network flow then splits the workload into simple- and complex-use-case traffic using *in-network processing* in the SDN. If the amount of traffic is larger than a single HWA resource instance can handle at most, further HWA resources may be provisioned that share the load (Figure 27c). From the perspective of the network flow, the data plane then acts not only as a splitter for simple- and complex-use-case traffic, but also as a load balancer between HWA resources.

State transfer

Seamless transition between these types of HWA resource usage however requires a non-trivial state transfer between commodity-CPU-based virtual machines and HWA resources, e.g. for session information (Chapter 4). This can result in a significant delay, which is added to provisioning and reconfiguration delay that may occur.

5.2 SYSTEM ARCHITECTURE

Assumptions

The proposed system architecture (Figure 28) builds up on an NFV architecture running on commodity hardware like x86-based CPUs, for example OpenStack. Aforementioned optimizations like SR-IOV or DPDK (Section 2.8.2) can be used to speed up commodity CPU network I/O. vNF software is still primarily shipped as code which can be executed by (virtualized) commodity CPUs.

Software structure

In the new architecture proposed, the vNF software additionally includes one or multiple packages with hardware behavior descriptions for the part of the network function that might be offloaded to hardware. The software may contain multiple

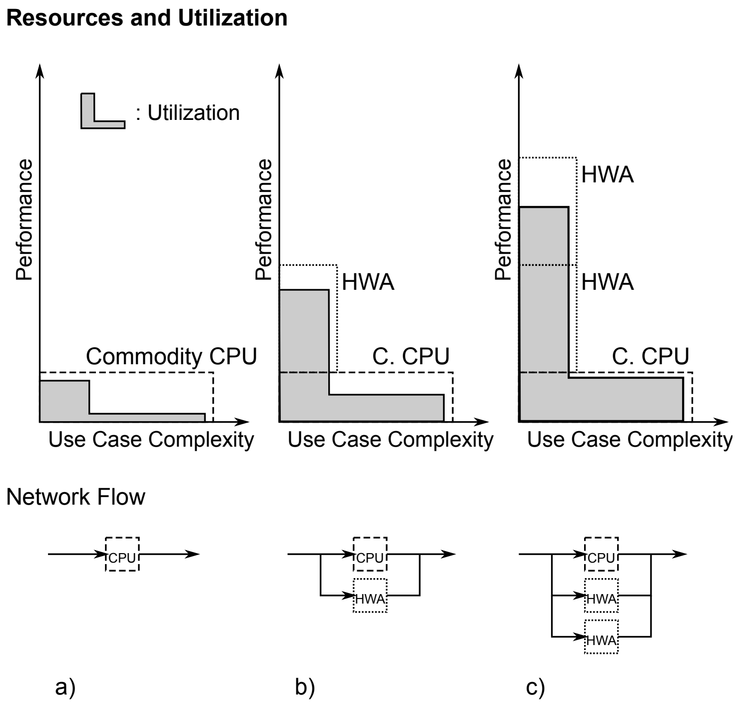


Figure 27: Simplified model of use-case-separated workload distributed to general purpose processors and HWA depending on current performance requirements. [122]

packages for the same behavior, but for different HWA types. For interoperability, preferably vendor-independent abstract language code should be used, although this requires compilation or synthesis. For instance, a PPPoE access concentrator may be shipped with a HDL description for hardware acceleration of session traffic, as well as pre-compiled code for different NPUs in case no compatible FPGA can be provided.

5.2.1 *HWA Modules*

A hardware acceleration (HWA) module consists of one or multiple HWA processors with dedicated network interfaces. Every HWA module is able to talk to the cloud platform through the HWA southbound interface. The primary tasks of such a module are to manage its HWA resources (①) and handle control-plane communication. These tasks may be fulfilled by the HWA processor itself or by using a general-purpose co-processor.

Physical manifestations of HWA module

An HWA module can manifest in various physical configurations: It may be a vNF hypervisor carrying multiple HWA processors (e.g. PCI cards). In this *collapsed* configuration, a vNF and its provisioned HWA resource may reside on the same physical node, but a vNF instance on a commodity CPU may also obtain an HWA resource provisioned on another hypervisor. Thus, hypervisors can share their HWA modules, depending on their load. Furthermore, a module can run in *standalone*

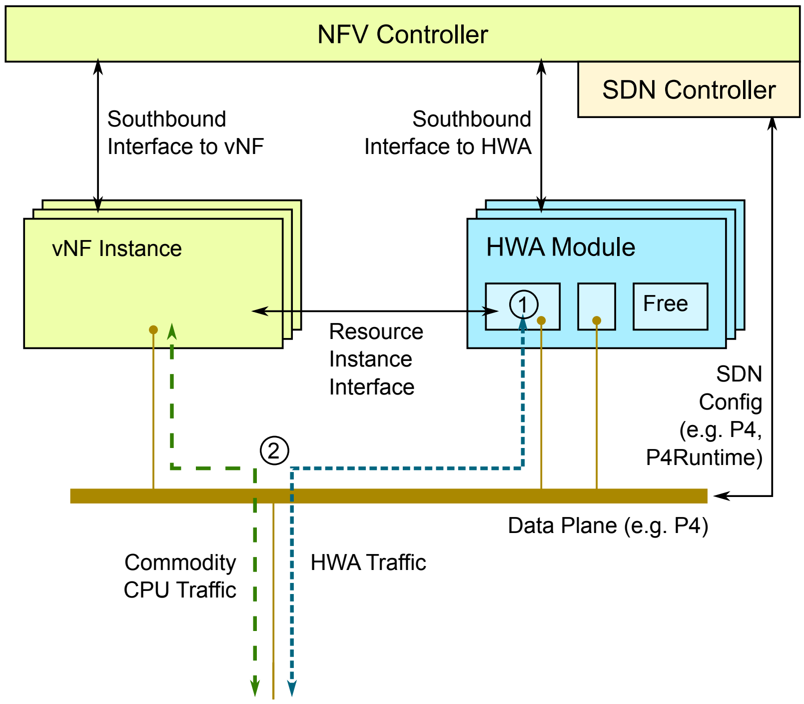


Figure 28: Architecture for elastic provisioning of HWA to vNFs. The vNFs are actually running on hypervisor nodes, which is not depicted here, as this does not play a role in our abstraction.[122]

configuration, this means as a physical node with the only purpose to fulfill the HWA module tasks. Moreover, an HWA module may be integrated directly into switches, reducing the number of cables involved.

5.2.2 *Interfaces*

The NFV infrastructure implements three new interfaces for HWA resource provisioning: the vNF and the HWA Southbound Interface, and the HWA Resource Instance Interface.

HWA south-bound interface

The **HWA south-bound interface** has two tasks. First, it allows HWA modules to register at the cloud platform with their resources, capabilities and network links. Thus, the platform has an overview of acceleration hardware available on the network. Secondly, the platform can use it to reserve HWA resources on the modules, and initiate communication between the vNF and the HWA module (through the HWA Resource Instance Interface).

vNF south-bound interface

The **vNF south-bound interface** is used by the commodity CPU vNF instance to acquire HWA resources. Therefore, the vNF gives a list of hardware descriptions available, while the platform checks available resources (FPGA gates, NPU cores, etc.) and may provide them in the specified amount. On success, the platform finally gives the vNF instance access to the HWA Resource Instance Interface to communicate with the HWA module. If the resource is not needed anymore, the vNF frees the resource by telling the platform. Moreover, the interface allows the vNF instance to tell the platform whether to direct traffic to a particular HWA instance provisioned, or not. Use case splitting (②) – like in-network processing – benefits from a data plane more flexible than OpenFlow, provided for example by the protocol-independent match/action instruction set of P4 [11]. To achieve this with OpenFlow, languages like Frenetic[1] or Pyretic may be used to specify and update these rules.

HWA resource instance interface

Access to the **HWA resource instance interface** is given to a vNF instance by the platform upon successful HWA resource assignment. The interface's traffic may be directly delivered to the HWA module by assigning a datapath between the vNF instance and the HWA module using the SDN. First, the interface is used to program the HWA resource with the compatible hardware behavior description. Secondly, the device is configured, and its configuration is continuously updated through this interface: For example, content-addressable memory (CAM) tables need to be regularly updated to reflect new sessions in an IPSec endpoint use case. Therefore, the HWA Resource Instance Interface may provide direct memory access to the vNF for table updates, but must care about the delay present on the interface's path through the SDN.

5.3 EVALUATION METHOD

Contribution Overview

In the last section, we have assumed that acceleration hardware is costly, and resource utilization can be increased, as well as costs can be decreased, by claiming hardware acceleration resources only on demand. This section provides an approach to *quantify* the assumed benefit of elastic hardware accelerator provisioning. Our contribution is three-fold:

- First, we have implemented a DPI network function on both an FPGA and a commodity CPU (Appendix A.2). Both implementation variants are evaluated regarding their performance and timing aspects (Section 5.4.1).
- Secondly, we propose a model to evaluate the benefits of elastic provisioning (1) compatible with *heterogeneous* computing resources (commodity CPUs and hardware accelerators like FPGAs), and with the ability (2) to derive cost savings, given a cost model (Section 5.3.2).
- Thirdly, we use the measured performance results, a resource demand model, and the determined costs of a commodity-CPU VM and an FPGA board as reference parameters for our evaluation model, and present and discuss the obtained quantified results (Section 5.4).

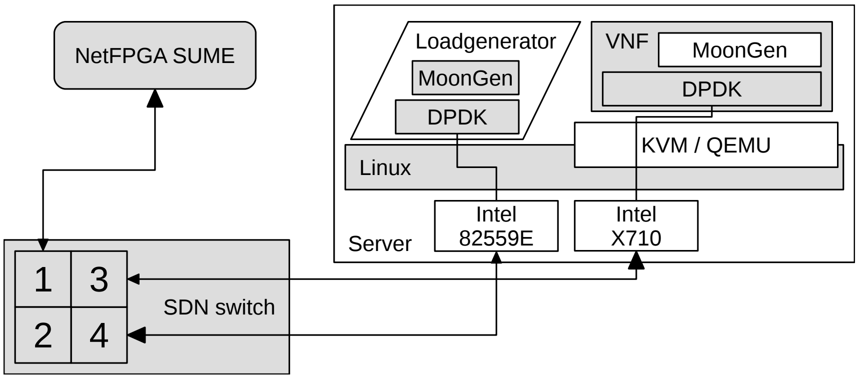


Figure 29: Testbed setup. [118, 144]

5.3.1 *Performance Evaluation Testbed*

Testbed overview

We use the following testbed to evaluate our DPI NF (Appendix A.2), implemented on both the commodity CPU and the FPGA, with determining the *throughput* and *latency* according to RFC 2544 for each of the implementations. Figure 29 sketches our testbed-based evaluation. For the experiment, we use a Dell R630 server being the hypervisor for the commodity CPU function, but also carrying the load generator. The dual-socket server is equipped with two 10-core Intel Xeon E5-2660 v3 CPUs, on which we have disabled HyperThreading for better poll-mode driver support in DPDK. On the server, the Ubuntu 15.10 operating system executes the KVM hypervisor, hosting the two aforementioned VMs. The VM and the load generator each exclusively accesses one of the 10 Gigabit Ethernet network interfaces, while the VM uses PCI-PT in order to reach highest performance, not deteriorated e.g. by vSwitches. Another computing node carries the *NetFPGA SUME* board, the node only needs low-performance CPU support, as it is only required for controlling the FPGA board and not involved in data plane processing. Our testbed furthermore comprises an AS5712-54x bare-metal switch. The switch runs the PicOS operating system in order to provide SDN and OpenFlow support, and interconnects all components in our lab.

Workload generator

The workload generator does not run in a VM, it runs directly on the host system. The MoonGen software [37] adheres to RFC 2544 and conducts a 3-phase measurement (throughput, latency and packet loss). Every experiment runs for 10 seconds, with packet sizes in classes from 64 bytes to 1518 bytes according to RFC 2544. For sub-millisecond latency measurements, the MoonGen software supports a hardware timestamping feature of the Intel 82599 network interface card, which we also use for our experiments. To ensure the load generator works as expected, we have successfully tested it in a loopback configuration first. Except for 64-byte packets (90,64%), we achieved a throughput of more than 99,99% of the available link capacity (10 Gbps).

1 http://www.frenetic-lang.org/

5.3.2 *Elasticity Evaluation Method*

Testbeds are limited in capacity, like in our performance evaluation, this problem is not given when using a model-based evaluation on pre-measured performance results of the processing units. Therefore, in this section, we describe an evaluation model to quantify the benefits of elastic provisioning which uses the performance evaluation results obtained with the testbed from Section 5.3.1 as *reference* parameters. However, to take into account that future NF implementations might provide different performance, we also discuss and present the effect of varying these parameters. As discussed in Section 3.2.1, Herbst et al. [66, 67] have suggested a metric for elasticity evaluation which has been adopted by the SPEC benchmarking standards. We adopt the terminology and some value primitives from the benchmark of Herbst et al., however, our evaluation model is different in the sense that it focuses on efficiency and costs. Furthermore, the model of Herbst et al. cannot be directly used for our elasticity evaluation because it must be adapted to support *heterogeneous* resources.

Assumptions

We assume that the provisioning process only has current or past information available for its decision whether more resources must be provisioned to meet a resource demand. It also cannot use prediction methods to forecast behavior. Furthermore, the process decides to increase provisioned commodity CPU or FPGA resources (depending on the scenario option) whenever it decides that resources become insufficient. Likewise, resources are *released* if less resources are required to handle the workload. We furthermore assume that the resources provided by a commodity CPU or an FPGA correspond to the capacity which the instance can process, determined by our performance evaluation. Thus, they can only be provided in discrete quantities described by Equation 14, where c_{ccpu} is the capacity of the commodity CPU and c_{fpga} the capacity of the FPGA. Even in an ideal provisioning system which can scale up and down at any time, the ability to provision as much resources as needed is restricted by the discrete provisioning values given by this equation.

$$\{n \cdot c_{ccpu} + m \cdot c_{fpga}; n, m \in \mathbb{N}\} \tag{14}$$

Scaling up and down

Figure 30 illustrates resource demand (dashed curve) and resource supply (solid curve) during a scale-up operation. The increase of resource demand results in a short period of **underprovisioning** (red area), as the resource supply cannot immediately follow the demand, which is caused by the setup time required by FPGA or commodity CPU instances (t_{fpga}, t_{ccpu}). After the new resources are set up, the system is subject to **overprovisioning** (light-brown area). When determining the underprovisioning, the different start-up times of the two resource types must be considered.

By summing up the positive areas of supply minus demand (brown areas) over any time interval, we obtain the total overprovisioning O, likewise, the sum of the negative areas is the underprovisioning U (red areas) over any time interval. Furthermore, we define the **timeshares**. ts_{fpga} corresponds to the total time any

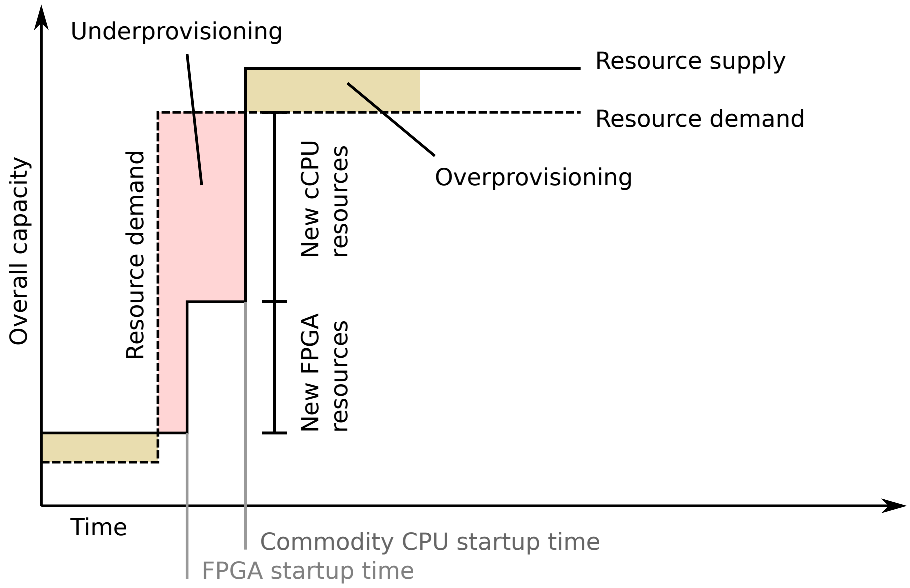


Figure 30: Scaling up, over- and underprovisioning.

FPGA instance has been in operation, and ts_{ccpu} for the commodity CPU instance respectively. For example, two FPGA instances running for 2 days results in $ts_{fpga} = 4days$.

Cost metric

Unlike the metric of Herbst et al. [67], we envision to also consider the *costs* of an infrastructure, which comprise two components: First, *underprovisioning* can cause unavailable service, for example it may lead to SLAs which have not been fulfilled. Secondly, a time-dependent *infrastructure cost* can be defined, which arises from operating the infrastructure. For the latter, we use a relative cost metric, normalized to the costs of an FPGA instance (a physical FPGA unit or a programmed region). The cost is composed of the sum of the timeshares of a specific instance type, while the commodity CPU is multiplied with the relative instance cost w_{ccpu} first (Equation 15).

$$\begin{aligned} cost = ts_{fpga_{base}} + w_{ccpu} \cdot ts_{ccpu}, \\ with \quad w_{ccpu} \in]0, 1]. \end{aligned} \tag{15}$$

The commodity CPU instance cost factor w_{ccpu} is the relative cost of operating an instance *per time* if the FPGA instance has the costs of 1 over this time. For example if an FPGA instance costs 100 cents per hour and a commodity CPU instance 40 cents per hour, $w_{ccpu} = 0.4$.

Workload dataset used

In our model, the resource demand is *sampled*, it is only determined in *discrete* time intervals $t_0, ...t_n$. We therefore consider a resource demand to remain constant during any sampling interval $[t_i, t_i + 1[$. We select workload data from Cortez et al. [28][2] for evaluation (Figure 31). The data represents a daily usage pattern of

2 CSV data obtained from https://datamarket.com/data/set/232g/

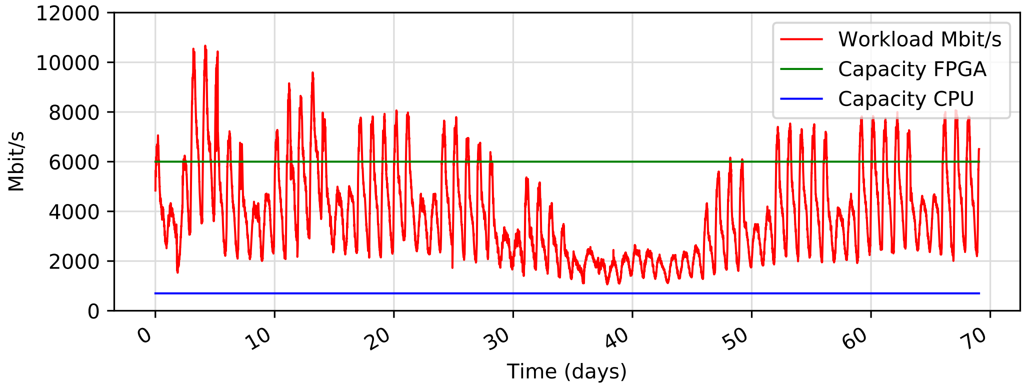


Figure 31: The workload pattern from Cortez et al. [28] which is used for evaluation. It lasts 69 days, including Christmas and New Year. It also depicts the default FPGA and commodity CPU capacity assumed in the evaluation.

a backbone of an academic network collected between November 19th, 2004, and January 27th, 2005. The data sampled in 5-minute intervals shows an oscillating load especially during workdays, furthermore workload is lower between Christmas and New Year (day 35-45). The original data ranges between 1 and 10 Tbit/s [28], thus we scaled it down by a factor of 10^3 so that it conforms to a smaller-scale NF scenario.

Evaluation Process

The evaluation process makes a resource allocation decision iteratively for every workload sample in the scenario. Therefore, a decision algorithm must specify a set of resources (of FPGA or Commodity CPU instance types) which are capable of processing the requested workload. As only two resource types are used in our model at maximum, the decision can be efficiently and cost-optimally solved with the algorithm in Appendix A.1.

The algorithm provides a number of commodity CPU and FPGA instances which must be provisioned, which is multiplied with the sample interval (5 minutes) to obtain the commodity CPU and FPGA timeshares of this period. The sum of the timeshares of all samples results in the total timeshare over the scenario which directly relates to costs. Furthermore, the process calculates the over- and underprovisioning for each sample. To calculate the underprovisioning due to startup times, the allocation decision of the last sample is consulted.

Scenario Options

To be able to compare the properties of different strategies, we specify and implement the following scenario options for evaluation:

- **Elastic/Static**: In the elastic case, the resources are elastically provisioned based on the workload of the current sample. As a new FPGA and commodity CPU require a certain startup time, underprovisioning may occur. In the static case, the resources are provisioned over the entire measurement period of our workload scenario (69 days). Therefore, the resource allocation is made once for the maximum workload in the scenario and held over the entire period.

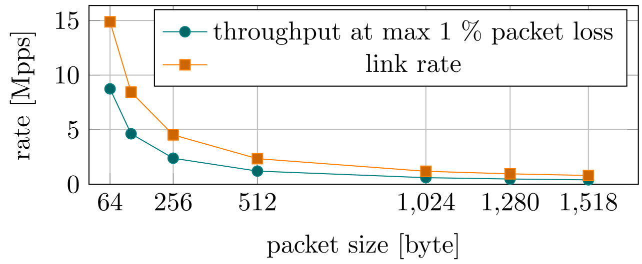


Figure 32: Throughput of the vNF instance on the FPGA. [118, 144]

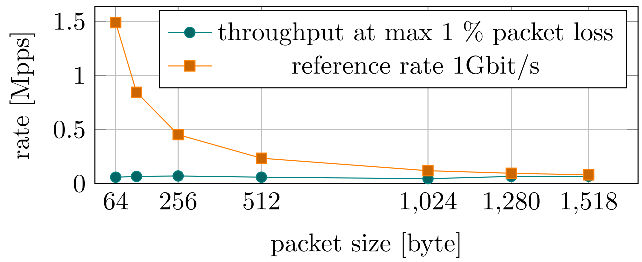


Figure 33: Throughput of the vNF instance on the commodity CPU. [118, 144]

- **Mixed/[Type]-only**: To fit a certain resource demand, the scenario mixes FPGA and commodity CPU instances in the instance pool if the *mixed* option is selected.
- **PerMbit**: By default, a commodity CPU instance causes fixed costs independently of its utilization. *PerMbit* is a special case where the commodity CPU resources only produce costs for the workload which they process. For example, if a CPU instance is only utilized by 40%, its costs are only 40% of a commodity CPU instance.

5.4 EVALUATION RESULTS

In the following section, we describe our evaluation results. First, we present and discuss the performance evaluation results of the NF implementations conducted for an FPGA and a commodity CPU. Secondly, we present a cost estimation based on market prices to obtain relative costs of provisioning an FPGA instance and a commodity CPU instance per time. The obtained performance and cost values are used as reference parameters for our elasticity evaluation model. The results of this model are finally presented.

5.4.1 *Performance*

Throughput measurement results

In Figure 32, we can observe the throughput measurement of the FPGA instance. They are depicted in *million packets per second (Mpps)*, but the link rate is bound to $10 Gbit/s$. Therefore, the link rate as the upper limit is also plotted in the corresponding Mpps value, in Figure 32, this is a reference rate of 1 Gbps instead. The

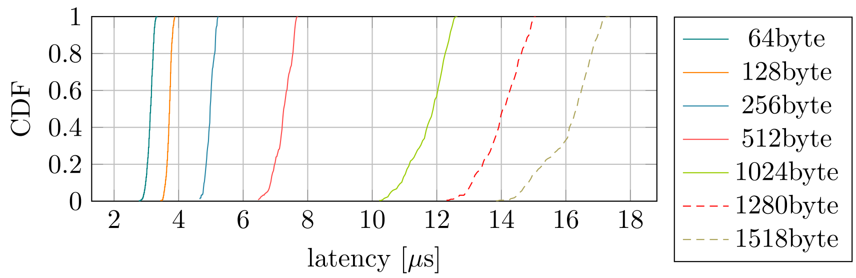


Figure 34: Latency of the vNF instance on the FPGA. [118, 144]

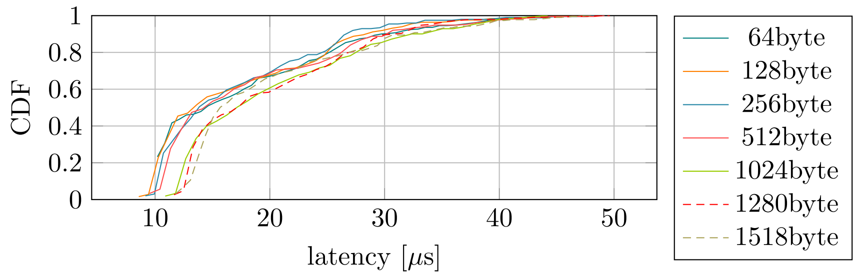


Figure 35: Latency of the vNF instance on the commodity CPU. [118, 144]

results show that the FPGA can reach around half of the link rate with every packet size. Comparing these results to Figure 35 where the results for the commodity CPU instance are plotted, we observe that the FPGA achieves great throughput for small packet sizes, for example 8.7 Mpps at a packet size of 64 bytes. The commodity CPU appears to achieve only very low packet rates around 0,06-0,07 Mpps, independently of the packet size, resulting in a peak of 840 Mbit/s for maximum-sized packets.

Delay measurement results

In Figure 34 and 35, the latency results are depicted for both instances using a cumulative distribution plot. The latency of the FPGA implementation ranges around $3\mu s$ and $17\mu s$ and appears to vary with packet size: small packets are forwarded very quickly, while larger packets take longer. Besides the packet size, the latencies do not vary much. Ragarding the commodity CPU implementation, there is a high variance in the delay performance. Most packets are only delayed up to $20\mu s$, but there is a tail of packets in the distribution needing up to $50\mu s$. Furthermore, even for small packet sizes, only a very small fraction of packets can achieve delays below $10\mu s$ unlike the FPGA implementation.

Startup times

In our testbed, we have observed that a virtual machine on a commodity CPU requires a startup time of approximately 12 seconds on average. The FPGA can be started up by programming it over the 32-bit SelectMAP interface, which operates at $100MHz$. Loading the bitfile with a size of $229Mbit$ to the FPGA results in a time of $72ms$ (Equation 16).

The very small time to instantiate the VM or an FPGA instance from a bitfile locally present does not include signalling and state transfer overhead required in a future cloud infrastructure, thus, we add 2 seconds for state transfer, which is

Instance type	Startup time	Workload Capacity	Relative Costs
Commodity CPU	12s	700 Mbit/s	0, 41
FPGA	2.1s	6 Gbit/s	1

Table 1: Summary of reference parameters obtained from the previous sections and used in our model.

in the order of magnitude of state transfer mechanisms like SliM and Duplication (Chapter 4). Conservatively, we set $t_{fpga} = 2.1s$ in our model.

$$\frac{229Mbit}{32bit \cdot 100MHz} \approx 0,072s \tag{16}$$

Packet sizes

The actual traffic in the Internet follows a bimodal distribution, where a large fraction of packets (44%) have a size of less than 100 bytes, only 12% have a size between 1300 and 1400 bytes, but 30% of the packets have the maximum size [113]. We assume a 1280-bytes packet size in our model. Assuming the determined peak rate of 0,07 Mpps for the commodity-CPU instance, we obtain a reference bandwidth of $716,8Mbit/s$ for a commodity-CPU instance which we round to $700Mbit/s$ (the remaining capacity can be seen as a buffer).

5.4.2 *Cost Estimation*

Cost weights

To determine the cost weights, it is necessary to make a market survey for both FPGA and commodity CPU resources for NFV. We could determine a monthly operational cost of 11,25 € (135,1 € per year) for a 4-core bare-metal server, which corresponds to a commercial offering of 11,99 € we have found. However, at the time of writing, we could not find market prices for cloud-based FPGA resources. Therefore, we estimate them based on hardware prices: The NetFPGA SUME retail price has been determined with 6256 €, and we assume that the card must be replaced every 5 years. We furthermore assume that the FPGA card can carry 4 NF instances simultaneously with the help of *Partial Reconfiguration* [169], and apply a model to consider energy costs [149]. This results in an instance's cost of 325 € per year. The corresponding cost weight of the commodity CPU is $W_{CCPU} = 0,41$. The cost values are an estimation only, may heavily vary depending on current market prices and infrastructure offerings, and thus should be calculated individually upon planning.

5.4.3 *Elasticity Evaluation*

In Table 1, we have summarized the reference parameters which we will apply to our elasticity evaluation. The parameters are partially estimated, as they highly depend on implementation details and market prices, and could be different especially due to future development. However, our evaluation also depicts interdependencies, so that results can be derived if any parameter deviates from these reference values.

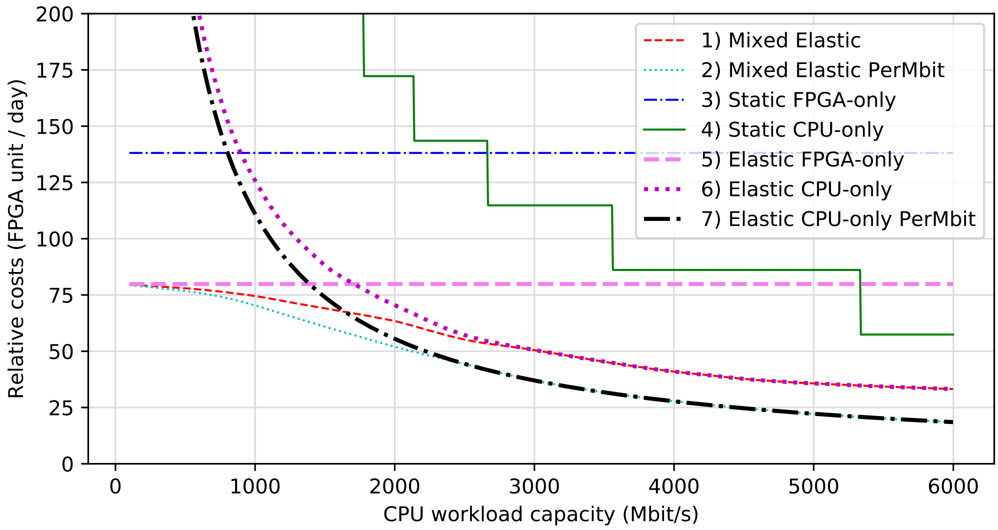


Figure 36: The relative costs of all provisioned resources, given in relative cost units (normalized to the costs of an FPGA instance). Depicted for different CPU workload capacity up to the capacity of the FPGA instance, other values according to our reference value set (Table 1).

Figure 36 shows the relative costs of all instances provisioned by our model during the 69 days of workload described by the sample set. The model uses different strategies to handle the requested workload which we have described with the aforementioned scenario options (Section 5.3.2), plotted with different lines.

As a first observation, we can see that approximately 40% of costs can be saved by elastically provisioning FPGA resources compared to static provisioning (Curve 5 vs Curve 3). These values are constant in the figure because CPUs are not involved, the values do not depend on CPU workload capacity, but on the reference parameters.

40% of cost reduction with elastic provisioning of FPGAs

Furthermore, we can see that via elastic provisioning of commodity CPU resources in a commodity-CPU-only environment we can save *more* than 50% compared to static provisioning of commodity CPU resources up to 3 Gbit/s (Curve 6 vs. Curve 4). At higher commodity CPU workload capacity values, the relative improvement decreases until it reaches the relative improvement of FPGA resources at 6 Gbit/s. We conclude that although we can highly reduce costs in FPGA-only environments through elasticity, CPU-only environments can save more costs because of elasticity. This is because while CPU instances provide a lower capacity than FPGA instances, they can be provisioned with a higher accuracy than FPGA resources (a more fine-grained provisioning is possible).

Larger relative cost reduction in classic environments

Besides the relative cost reduction for each instance type, we can observe that the elastic provisioning of FPGA resources provides lower costs than an elastic provisioning of commodity CPU resources up to a commodity CPU workload capacity of 1700 Mbit/s (Curve 5 vs Curve 6). Naïvely, a CPU should be only more cost-efficient than an FPGA if its workload capacity is larger than the FPGA workload capacity times the CPU's relative costs, this is a capacity of larger than

Elastic FPGA vs. elastic CPU

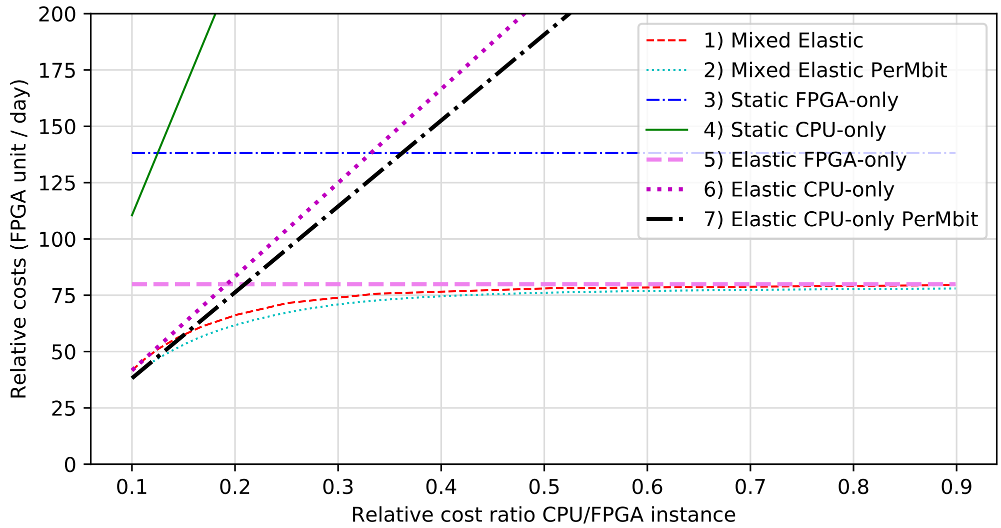


Figure 37: The relative costs of all provisioned resources, given in relative cost units (normalized to the costs of an FPGA instance). Depicted for different CPU workload capacity up to the capacity of the FPGA instance, other parameters according to our reference parameter set (Table 1).

2460 Mbit/s in our scenario. However, this only applies if the FPGA can always be fully utilized. Due to the higher accuracy of low-capacity CPU instances, which results in smaller overprovisioning, the cost advantage of commodity CPUs is already located at 1700 Mbit/s.

Mixing FPGA and commodity CPU instances

In another scenario, we mix FPGA and commodity CPU instances in order to both achieve a high accuracy by provisioning commodity CPUs and to harness a potentially higher performance/cost ratio of FPGAs (Curve 1). Here, we obtain an additional cost reduction of approximately 15% at 1700 Mbit/s (Curve 1 vs Curve 5 or 6). We also consider a model where costs for CPU resources are only accounted for workload they actually process (PerMbit scenario option, Curve 7). Here, an optimal fitting can be achieved with no overprovisioning.

Figure 37 depicts the effect of varying the relative costs of a CPU instance, assuming the reference workload capacity of 700 Mbit/s. Except for the *mixed* scenario options, the effects are linear, with a clear cost advantage of the FPGAs beyond 0,2. At 0,2, the advantage of the *mixed* mode maximizes with a cost reduction of approximately 15% compared to the single-type elastic modes.

Figure 38 shows how much bandwidth capacity is overprovisioned for each scenario option. The *static* scenario options (Curve 3 and 4) obviously suffer from the largest amount of overprovisioning. The static CPU-only overprovisioning follows a sawtooth curve: The overprovisioning increases due to larger capacity, but at certain steps, an entire instance can be saved after a certain performance increase. The overprovisioning of the mixed-elastic scenario (Curve 1) mostly follows the elastic commodity CPU scenario, except that with low CPU workload capacity

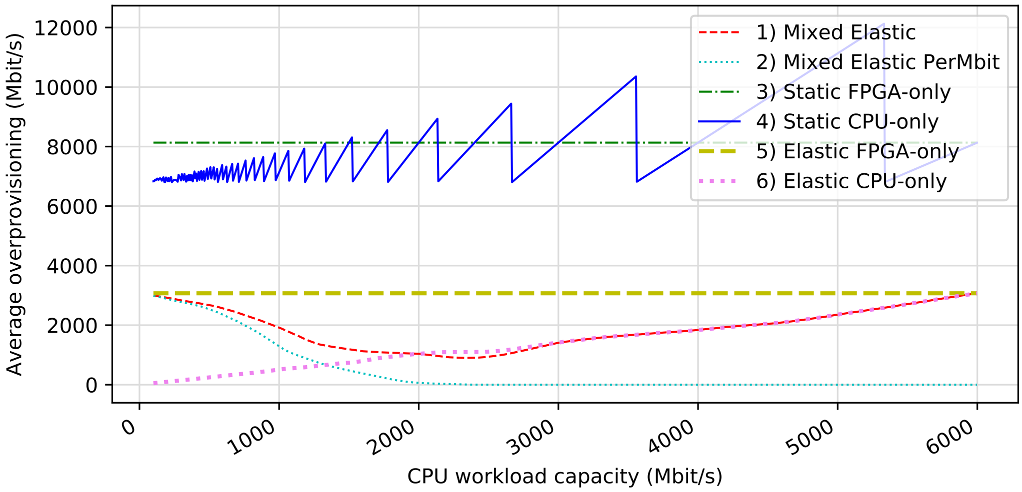


Figure 38: The average overprovisioning, given in megabits per second. Depicted for different CPU workload capacity up to the capacity of the FPGA instance, other parameters according to our reference parameter set (Table 1).

(under 2000 Mbit/s), the mixed-mode placement strategy decides that an FPGA is more efficient, even when taking into account a higher overprovisioning.

In Figure 39, the timeshares (in days) are depicted for various scenario options (for several scenario options these are splitted into the two instance types). In scenarios with the *mixed* option (Curves 1, 2 and 7, 8) and with increasing CPU workload capacity, the affinity to provision FPGA instances decreases, while more commodity CPU instances are provisioned. The *Mixed PerMbit* scenario (Curves 7 and 8) abruptly stops using any FPGA resources at 2460 Mbit/s (the FPGA workload capacity times its relative costs), Here, the placement decides that FPGA resources are inefficient in any case. However, the default scenario (Curves 1 and 2) provisioning of FPGA instances entirely stops using FPGA resources later (at 3000 Mbit/s) with a smooth transition, because in a few special placement cases, FPGA instances can be used to avoid CPU instance overprovisioning.

Costs can also result from underprovisioning [67], where insufficient resources are available to handle the requested workload. In Figure 40, we can see that especially CPU resources suffer from underprovisioning[3], and with increasing CPU workload capacity, the underprovisioning timeshare decreases. The reason is that the higher accuracy of low-capacity commodity CPU instances, which can be provisioned on a more fine-grained level, also leads to frequent underprovisioning. In contrast, FPGAs provide more capacity which can be used as a buffer to avoid future underprovisioning.

Finally, Figure 41 shows the fraction of traffic which cannot be processed due to underprovisioning. In the elastic commodity CPU scenario, around 0,01% of work-

3 The larger setup time of the CPU instances according to our reference parameters does not explain the timeshare larger than two orders of magnitude.

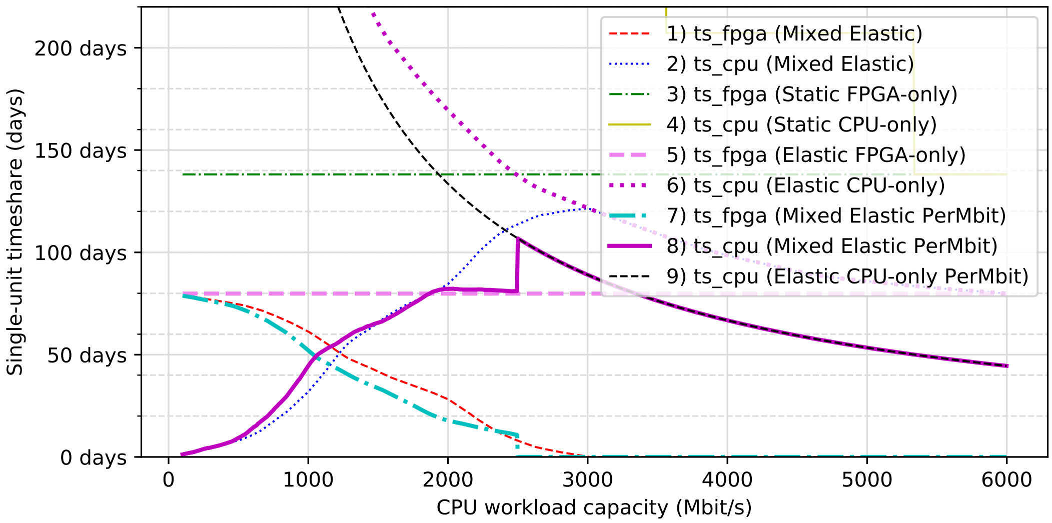


Figure 39: The timeshares of the instances (ts_fpga, ts_cpu) in days. Depicted for different CPU workload capacity up to the capacity of the FPGA instance, other parameters according to our reference parameter set (Table 1).

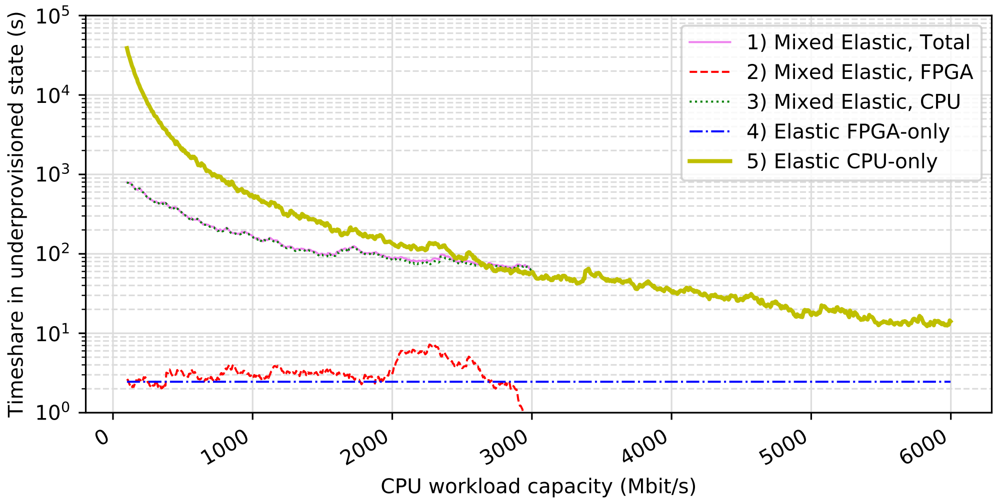


Figure 40: The timeshare of an instance type being in an underprovisioned state, given in seconds. Depicted for different CPU workload capacity up to the capacity of the FPGA instance, other parameters according to our reference parameter set (Table 1).

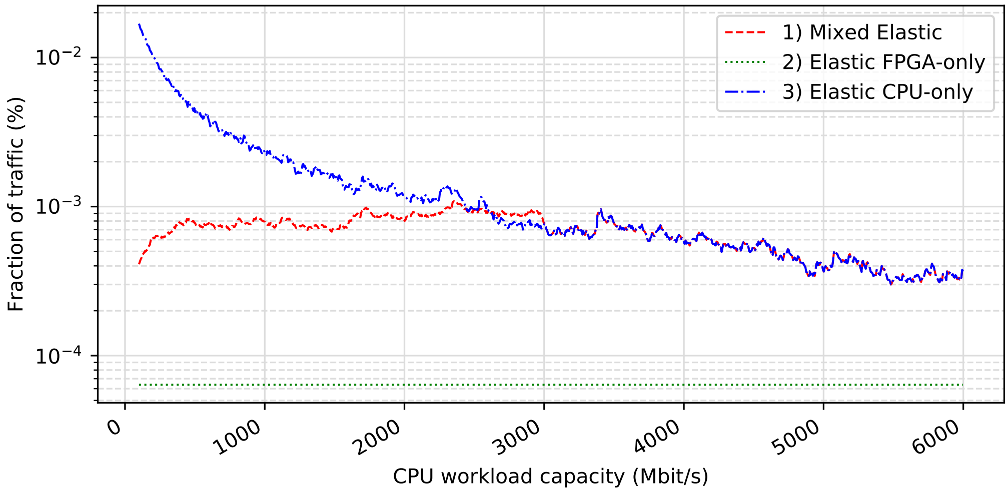


Figure 41: The fraction of traffic which cannot be processed because of underprovisioning, given in percent. Depicted for different CPU workload capacity up to the capacity of the FPGA instance, other parameters according to our reference parameter set (Table 1).

load suffers from underprovisioning for small CPU workload capacity, while only a very little underprovisioning is caused by the FPGA-only scenario. Furthermore, the FPGA instances in the mixed-elastic scenario keep the fraction of traffic suffering from underprovisioning below 0,001% even for low CPU workload capacities.

5.4.4 *Limitations of the Model*

The model is subject to several limitations. First, it assumes the provisioning operates in a *reactive* way. This means that it only follows service demand requests, however, prediction mechanisms could forecast future demand and thus anticipate underprovisioning, for example by booting up instances in advance. Furthermore, the model uses megabits per second (Mbit/s) at the current time to measure resource demand. For other NFs primarily involved in header processing, the bottleneck is more accurately measured in megapackets per second (Mpps), the suggested model could however be quickly adapted to this metric. Finally, the model does not currently distinguish between simple- and complex-use-case traffic like suggested in the split-architecture we have proposed (Section 5.2). The benefits of the resource allocation for simple- and complex-use-case traffic should each be evaluated in a separate instance of the model.

5.5 SUMMARY

In this chapter, we have suggested to close an elasticity gap regarding hardware acceleration. We have therefore proposed an architecture which allows us to acquire heterogeneous processing resources on demand, available from a resource pool over a network. The presented architecture of elastic provisioning must however be justified by a benefit. Therefore, we have implemented a DPI NF on both an FPGA and a commodity CPU and measured its performance characteristics in a testbed. Furthermore, we have specified a model based on the terminology of Herbst et al. [66, 67] to evaluate these benefits. The model has been supplied with the obtained performance data of our NF implementations, an estimation of the relative costs of provisioning FPGA and commodity CPU hardware per time, and a traffic measurement from a university backbone [28] representing a daily usage pattern which we use as a workload pattern to define the resource demand.

RQ 2.1

We can conclude the evaluation of elasticity benefits with the following key findings:

RQ 2.2

- We could save approximately 40% of costs by elastic provisioning of FPGAs, compared to static FPGA provisioning.
- We could save approximately another 15% of costs compared to elastic provisioning of FPGAs if we use a set of mixed FPGA/commodity CPU instance resources to fill an elasticity gap.
- The risk of underprovisioning due to startup times is significantly lower when using FPGA instances, either in mixed infrastructures or in FPGA-only ones. This is partially caused by startup times of FPGAs which we have found to be lower, but primarily because their larger capacity results in more capacity reserves, thus only causes infrequent requests to provision more resources.
- The cost benefit of elastic provisioning of FPGA resources is slightly lower than the benefit of elastically provisioning commodity CPU resources, as the large capacity of FPGA resources results in a larger and more costly overprovisioning.

6 LEVERAGING MERCHANT SILICON IN BARE-METAL SWITCHES

The ideas and findings presented in this section have been previously published by the author of this thesis [119]. Several sections contain citations from the author's publication [119] which are not explicitly marked. For more information, refer to Appendix B. Cited figures are explicitly marked.

A core part of the NFV architecture concept is the usage of *commodity hardware* to implement NFs, primarily in order to save costs. NFV hardware mostly comprises commodity CPUs, however, performance shortcomings of CPUs and virtualization (Section 2.8) constitute a drawback when using NFV where high performance is needed. Chapter 5 investigated solutions based on programmable hardware acceleration like FPGAs.

Approach

A different approach is to consider the forwarding ASICs of bare-metal switches (Section 2.11) as commodity hardware and to include it in the NFV ecosystem. A vendor lock-in into proprietary interfaces of these forwarding ASICs is anticipated by the wide availability of some vendors' chipsets and their open middleware APIs [17], in industry, these chipsets are also known as **merchant switch silicon**. While many previous works have used bare-metal switches in a software-defined infrastructure for *forwarding* only [53, 92, 162, 163], we consider our work to be the first to implement a BRAS entirely on one of these devices. This carrier-grade NF aggregates thousands of subscribers and has highest scalability and throughput performance demands, which can be better satisfied by the 720 Gigabit-per-second backplane capacity of the *Edge-Core AS5712-54x* bare-metal switch in use, but not by commodity-CPU NFV platforms [105] of similar rackmount size or energy consumption.

Advantages

With bare-metal switches, it is feasible to operate the control plane and data plane on the same node, while the control plane software is still decoupled from the data plane over an API and can be freely customized or exchanged. Compared to remote control planes connected via OpenFlow, this has several advantages like higher fault tolerance and lower complexity (Section 2.11.1), furthermore, it reduces latency and offloads the controller network, as several controller tasks, like processing packet-in events and reacting on them, can be carried out locally on the device.

Limitations

The high performance, however, is restricted by the functionality of the forwarding ASICs. It is not possible to implement arbitrary NFs on their data plane, and it must be checked for every use case whether the data plane capabilities are sufficient. A further drawback is that elasticity and flexibility through virtualization – the "other" core part of NFV – is not inherently present on commodity ASICs.

Chapter Overview

Nevertheless, the performance advantage might outweigh aforementioned limitations, this is why we decided to analyze the capabilities of the switch to implement a BRAS. In Section 6.1, we present our implementation approach and describe how

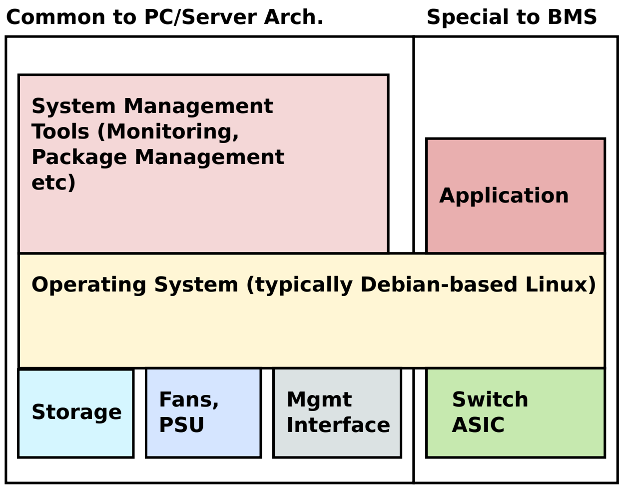


Figure 42: Comparison of a bare-metal switch architecture and a commodity PC/server architecture. [119]

we use the bare-metal system on a high level. Section 6.2 details the implementation and the silicon interface usage. We finally present our *verification* results in Section 6.3. As the load generators we had available during evaluation were not capable to reach both throughput and latency limits of the switch platform, we have restricted our evaluation to qualitative functionality tests.

6.1 APPROACH

Architecture Comparison

Figure 42 depicts a generic architecture of a bare-metal switch. At a first glance, the switch has an architecture which is very common to a commodity server, and from a management perspective, a bare-metal switch can be operated exactly as such: When installing a Linux-based OS (like Open Network Linux[1]) and connecting through the management interface, the switch will behave like a default Linux host[2]. From here, it is possible to install, compile and run Linux software as needed for own purposes.

Switch ASICs

The main difference of using a bare-metal switch compared to using a server is to get the *switchports* running, which is not possible with an out-of-the-box Linux so far. The switch ports will not be visible to Linux as network interfaces, this only applies to the management interface. Instead of appearing as individual devices to the Linux kernel, the switchports are tied together by a high-performance ASIC, providing a single interface to the Linux kernel only. The operating system and the application commonly does not receive and send packets over this interface, instead, it instructs the ASIC with rules on what to do with any packets coming in (e.g. to modify them), so called *forwarding rules*. The ASIC is capable of executing

1 http://opennetlinux.org/
2 An exception is that the bare-metal switches commonly do not have a VGA port and must be initially configured via the serial interface until SSH access is established.

these rules on packets with a very high data rate compared to CPU-based I/O. Nevertheless, ASICs are also capable of accepting rules to directly forward packets to the control plane, or send packets from it, however this should be rarely used in order not to degrade performance.

OS Alternatives

To avoid the complexity of data plane programmability, ASIC vendors have joined the bare-metal switch community and provide drivers, interfaces and abstractions to their data plane [17]. OS vendors for bare-metal switches also provide alternatives to the low-level ASIC programming APIs with middlewares for OpenFlow [132], Linux interfaces and bridges, or even a traditional command line, however, with additional license costs.

6.1.1 *Usage of OF-DPA*

As a major provider of switching ASICs, Broadcom has stepped up to the bare-metal switch community by supplying interfaces for ASIC programmability. The specification and libraries OpenNSL and OF-DPA are publicly available on GitHub [16]. They both consist of drivers, a daemon, and interfaces. OpenNSL originally focuses on traditional network management commands. In particular, typical API actions in OpenNSL are about adding/removing VLANs and ports to VLANs, switching ports to L2/L3 mode, adding/removing routes, port mirroring, QoS, statistics, link aggregation, and Virtual Extensible LAN (VXLAN).

OF-DPA Overview

The OF-DPA is an API to modify the forwarding behavior of the ASIC in a flow table format. The operations like adding and removing flow tables, group actions, and meters, are very similar to OpenFlow, however, the table types of its multi-table pipeline are very restricted in their functionality. Since June 2016, the OF-DPA APIs have become part of the OpenNSL library.

Restrictions

Like mentioned before, the similarity to OpenFlow allows the API to be directly accessed over this network protocol with an appropriate OpenFlow agent like *Indigo*[3]. However, because of the very restricted set of possible matchers and actions in every table, the semantics of an OpenFlow controller must be adapted to the particular table type restrictions, which contradicts the idea of OpenFlow's hardware independence. The detailed OF-DPA specification is available on GitHub [17].

Accessing the API

OF-DPA can be used either as an API by a local controller on the switch, or via OpenFlow by a remote controller. Regarding the first variant where we will focus on, a controller application can load a shared library, recommended for C and C++ development. For Python developers, a Python wrapper is also supplied with OF-DPA [18]. We have first tested the Python wrapper, which is stable and relatively easy to use, however, we experienced performance problems with controller packet-in/out operations, thus we have switched to the C header files. At the time of writing, we could not find any OF-DPA-based open-source software which uses the local API and which could be used as an example for using OF-DPA as a local API. Although Python example scripts are provided for a small set of operations, larger projects written in the C language could not be discovered.

Tables and group actions

3 http://www.projectfloodlight.org/indigo/

The OF-DPA pipeline [17] consists of two main entities, *tables* and *group actions*, which are both known from OpenFlow. Tables match a selected set of fields and can apply a selected set of actions on the packet and on several metadata fields, while group actions are a set of actions to be applied to a packet. Entries of several table types in OF-DPA can set a group action on a packet, which is only executed after the end of the ingress pipeline. This means a subsequent ingress table entry can also clear group actions previous tables have applied, which will lead to no effect of the group action. OF-DPA has very strict requirements on the sequence of tables which must be applied.

Types required by our use case

Although OF-DPA comprises a large number of table and group action types (some of them are for multi-protocol label switching (MPLS) termination/initiation and for VXLAN support), most VLAN-based L2/L3 use cases only require a small excerpt of it. This is caused by the fact that most tables have a built-in default action which forwards a packet to the next relevant table. In this thesis, we therefore restrict our explanation to the likely most required ones: the **VLAN** table, the **Policy access control list (ACL)** flow table, the *L2 Rewrite* and the *L2 Interface* group action. With these elements alone, it is possible to implement a Layer 3 hop with multi-field flow matching.

- The **VLAN** table matches the input port and the (first) VLAN tag only. If no entry exists in the VLAN table, a packet is dropped, which constitutes a VLAN filter per port. Except in special cases like removing a second VLAN tag, VXLAN or MPLS L2 initiation, the successor of the VLAN table is the Termination MAC flow table.

- The **Policy ACL** table can be seen as most powerful one: It supports wide-field *matching* on most packet headers, comparable to current OpenFlow versions. It is also possible to apply meters here. However, instead of applying versatile actions as in OpenFlow, the table is restricted to applying only the following *group actions*.

- An **L2 Rewrite group action** is applied by the Policy ACL table. It can rewrite the *VLAN ID* and the source and destination *MAC addresses* (note that IP rewriting is not possible in the OF-DPA version 2.01). The L2 Rewrite group action must apply an L2 Interface group action afterwards.

- The **L2 Interface** group action can be applied either directly or via the L2 Rewrite group action. It is just defined as a tuple consisting of an *output interface* and *output VLAN*. When applied, the packet will be sent out on the respective port tagged with the given VLAN. If the packet's VLAN does not match the one in the action, the packet is dropped (therefore, the L2 Interface group action is a kind of *VLAN filter*).

6.1.2 *BRAS NF Design*

A broadband remote access server (BRAS) is a fundamental network function in an carrier's network. The function terminates DSL access multiplexers (DSLAMs) from the subscriber side and provides access to the IP network. In some networks, the BRAS uses PPPoE at the subscriber's side to terminate multiple subscribers, in other networks, VLAN-based subscriber termination is used where we will focus on. A BRAS has strong requirements on performance, as it terminates and services a very large number of subscribers. Therefore, the BRAS network function is commonly implemented in rather costly appliances with special ASIC support.

Ports and addressing

Our BRAS has two internally-configured MAC addresses, the *subscriber-facing* and the *core-facing* one. Several ports are being designated for termination of the subscriber's side via a DSLAM or an optical line termination (OLT). Besides the subscriber-facing ports, *core* ports can be configured on the switch. These ports support a simple IP/Ethernet format and thus can be attached to routers in the carrier's core network.

BRAS Functions

In the upstream direction, the data plane must detect and redirect subscriber authentication attempts and control packets to the CPU. The BRAS must furthermore ensure that only authenticated subscribers (identified by their VLAN ID) can send packets originating from their designated IP addresses, obtained from a preconfigured lease pool. A three-color-based *meter* must be applied[4] to ensure subscribers can only consume the bandwidth of the purchased service. In the downstream direction, the BRAS must look up the packet's destination IP address, and forward it to the subscriber's VLAN ID if it belongs to an authenticated subscriber.

Additionally, a subscriber is assigned to a set of services (like Internet, VoIP and IPTV), which are either specified by an additional inner VLAN tag, or by the IP subnet which is used. Upon authentication, the BRAS terminates all services the subscriber is assigned to, and forwards it to a specified core port or VLAN.

6.2 IMPLEMENTATION

Our implementation of the aforementioned BRAS is planned to be realized with as much free software as possible. Therefore, we have selected *Open Network Linux* as the operating system platform (Figure 43). For performance reasons, the BRAS control plane is written entirely in the C language (depicted in blue). However, proprietary components cannot be avoided, as we require them for ASIC support. These proprietary components include the OF-DPA binary and the OF-DPA RPC client (depicted in red). As the BRAS control plane runs locally, we have decided to use OF-DPA without an OpenFlow agent and directly link against the OF-DPA API.

Startup

The BRAS software is loaded with an initial configuration file (see Listing A.2 in the Appendix for an example). The values in this configuration file are related to the supported services, subnets, core ports and MAC addresses, and cannot be changed at runtime. Upon startup, initial flow rules are installed (Section 6.2.2) and a Websockets server is started to accept subscriber configuration at runtime.

Subscriber state model

Figure 44 depicts the subscriber state model. Initially, a subscriber is not existing (white) in the device configuration. If a subscriber is added to the device via the Websockets connection, the subscriber is *initialized* (red): The *local state manager*

4 RFC 2697, 2698 or 4115

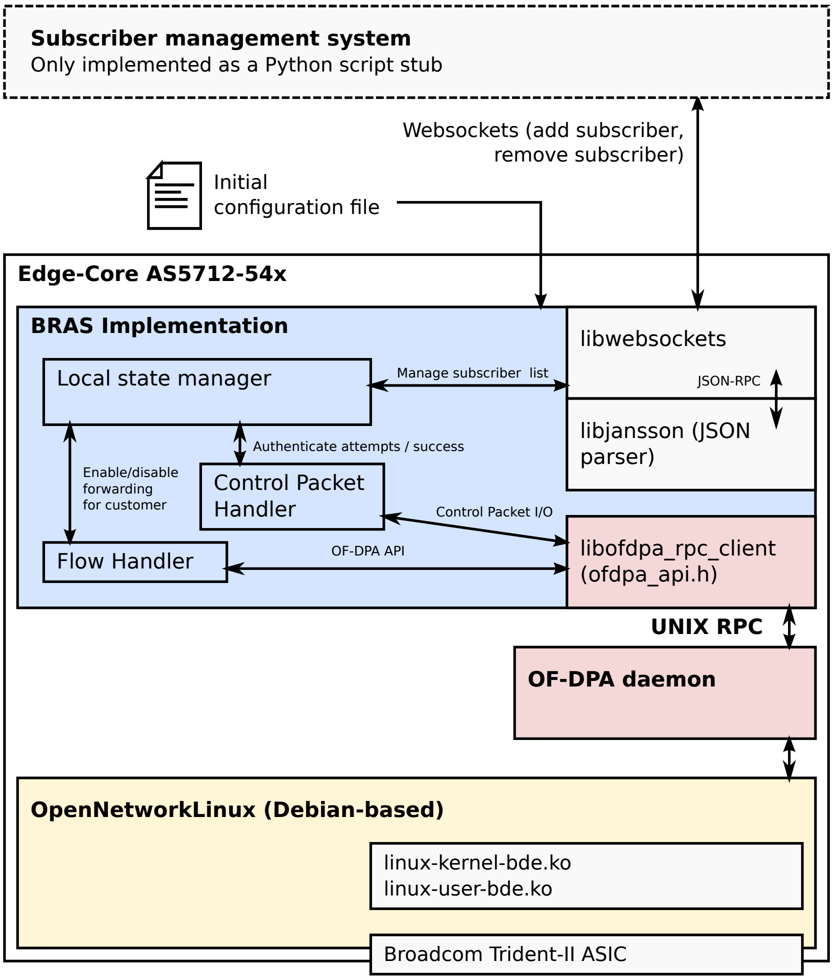


Figure 43: System architecture of the BRAS implementation [119]

installs flow rules which only forward authentication packets from the subscriber to be passed to the controller, however no forwarding is yet enabled. The *Control Packet Handler* then waits for authentication attempts of the subscriber, and if the authentication has been successful, the forwarding state for all the subscriber's services to the core network and back is established, the subscriber is then *authenticated* (blue). In the other direction, a subscriber becomes deauthenticated by not refreshing the authentication state in a pre-defined interval. Finally, a subscriber can be removed from the local state (e.g. by ending the service contract or moving to another BRAS region).

6.2.1 *Subscriber Management*

External interface

To initialize subscribers, a Websocket service is started which is accessible on the BRAS management port; subscribers can be added and removed by connecting to this service. Listing A.3 (Appendix) shows how a subscriber with a specified username and password is added in a JSON format. The specified OLT port and

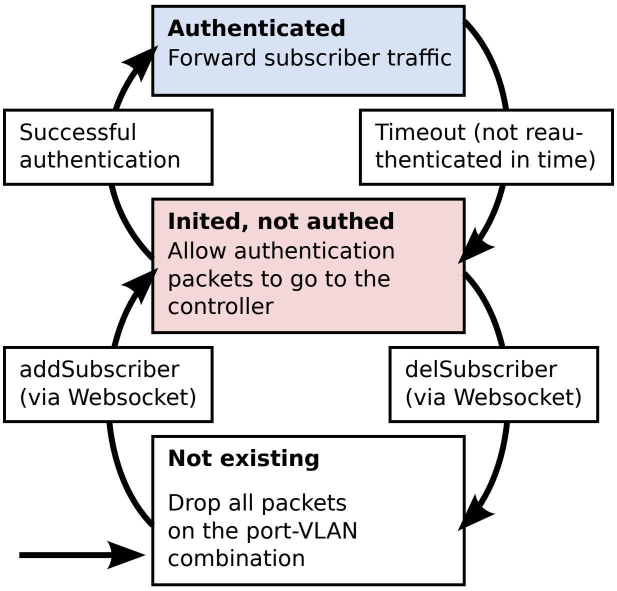


Figure 44: Per-subscriber state diagram in the BRAS switch [119]

the VLAN ID is expected when the subscriber authenticates. Furthermore, the subscriber is restricted to the given service VLANs. The subscriber ID is only for internal use (e.g. to be able to delete the subscriber when needed) and can be arbitrarily chosen.

Authentication method

In a BRAS for carrier-grade productive usage, standard formats for authentication should be followed. A candidate for a state-of-the-art standardized authentication protocol is the Extensible Authentication Protocol over LAN (EAPOL), which is part of the Institute of Electrical and Electronics Engineers (IEEE) 802.1X authentication standard. Subscribers requesting authentication can use this protocol via a Ethernet-based message exchange in conjunction with a Remote Authentication Dial-In User Service (RADIUS) server in the provider's backend. However, authentication is a task conducted by the control plane, the data plane's task is only to forward authentication packets to the controller (or the RADIUS server). The implementation of an EAPOL support is hard, to simplify the proof-of-concept implementation targeted to assess ASIC programmability for the data plane, we have designed a simple, light-weight authentication protocol. It is plain-text username-password-based, and therefore not secure especially against eavesdropping. We argue that to extend the proof of concept to EAPOL support, only the C-based control plane code must be extended, but not the data plane capabilities which we want to evaluate.

Authentication protocol

Figure 45 shows the packet format for the authentication request and response. Authentication requests (initially sent by the client) are always using the broadcast address on the Ethernet layer, as the destination MAC address is assumed to be previously unknown. The source MAC address is very important, as, upon successful authentication, it is used as the future customer premises equipment (CPE) WAN interface address (the address of the subscriber's home router). This is the only address allowed to send packets through the BRAS data plane, and

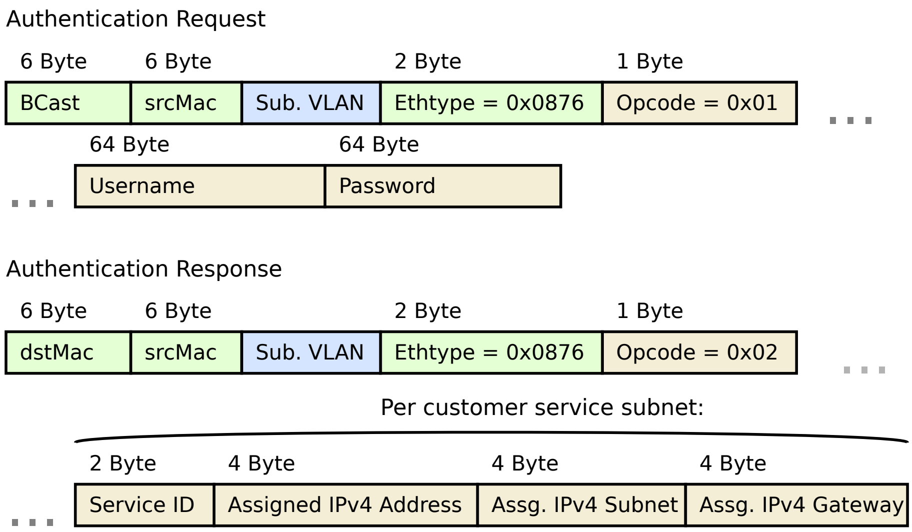


Figure 45: BRAS Implementation – Authentication Packet Format. [119]

this is the target MAC address for any packets sent to the subscriber. Next, the subscriber's VLAN is expected (assumed to be assigned by the DSLAM, which is agnostic of any authentication). The ethertype of the next header of the VLAN tag is set to 0x0876 which we have chosen as the ethertype of the custom protocol. After the opcode (0x01), the username and password fields are both of a 64-byte fixed size, containing null-terminated strings.

In the response packet, the destination MAC address contains the CPE which made the request, and the interior BRAS MAC address as the source, and the requesting subscriber's VLAN. The ethertype is equal to the request, but the opcode of the response is 0x02 upon authentication success, 0x03 otherwise. Upon success, for every service available, the service VLAN ID (2 byte), the assigned IPv4 address, the subnet mask, and the default gateway are provided[5].

6.2.2 *OF-DPA Dataplane Flow Model*

In this section, we describe the data plane implementation in OF-DPA 2.01. To a certain extent, the data plane follows the behavior of a typical Layer 3 hop (a router), which is straightforward to implement in OF-DPA. However, the greatest challenges for a scalable implementation in OF-DPA are beyond these functionalities, like the *antispoofing*, which requires a large table for source IP matching, as well as a fine-grained *metering* support. In the following, we describe two data plane modes, the *double-tagged* and the *single-tagged* mode. The latter mode was introduced, as the implementation of the double-tagged mode was not supported by the present OF-DPA API version.

Subscriber to core, double-tagged

Figure 54 (Appendix) depicts the flow model we have used for the double-tagged mode in the packet flow direction from the subscriber to the core (see Figure 46 for

5 We implemented the IPv6 data plane, but we did not implement IPv6 control plane functionality.

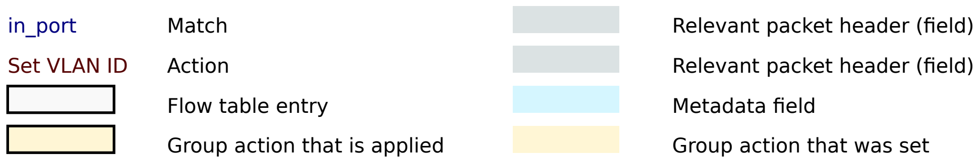


Figure 46: Meaning of the elements used in the following OF-DPA flow model figures. [119]

the meaning of the symbols). The L2 Rewrite and L2 Interface group actions have to be instantiated only once per service in this direction, the VLAN table entry must be instantiated per subscriber, while a VLAN 1 and the Policy ACL entry must be instantiated for every subscriber times every service. The handling of the ingress double tag was supported and successfully tested in this direction. In the pipeline, we introduce the term *ASIC VLAN* as an only internally-used VLAN ID, carrying information which would otherwise be lost between the tables due to the stripping of the outer VLAN tag. The ID can be arbitrarily chosen, however must be unique on the ingress port to be able to uniquely match the packet in the subsequent Policy ACL table. As the number of different VLAN IDs is 4096 (or a little bit less), the maximum number of subscribers times services on every port is restricted to this number.

Core to subscriber, double-tagged

Figure 55 (Appendix) shows the double-tagging mode in the direction from the core to the subscriber. Despite the support of a large number of entries to match the destination IP address in the Unicast Routing flow table, we had to use the Policy ACL table to be able to apply per-subscriber metering. The pipeline described here is not valid, as an Egress VLAN table entry, which is required for the Egress VLAN 1 table to add a second tag, cannot follow an L2 Interface group action. The Egress VLAN table is only accepted after an L2 Unfiltered Interface group action (here, the metadata field `ALLOW_VLAN_TRANSLATION` is set to 1), which itself can only be used in MPLS intiation or termination use cases.

Subscriber to core, single-tagged

Figure 47 shows the flow model for the *single-tagged* mode in the packet flow direction from the subscriber to the core. The only difference is that no service VLAN tag is used, instead, the service is identified by the source IP subnet of the subscriber. This approach is functional, provides a comparable isolation, and does not impose the scalability restrictions of the ASIC VLAN approach, leading to only 4096 subscribers times services.

Core to subscriber, single-tagged

In the packet flow direction of the *single-tagged* mode from the core to the subscriber (Figure 48), no egress tables are required, thus, the pipeline of this mode is functional. Like in the double-tagged mode, the Policy ACL table must be used to be able to apply a per-subscriber meter, the large Unicast Routing table cannot be exploited.

6.3 EVALUATION RESULTS

Test setup

Figure 49 depicts the test setup we have used to evaluate correct operation of the BRAS proof of concept. Therefore, we have connected a test server to the switch with

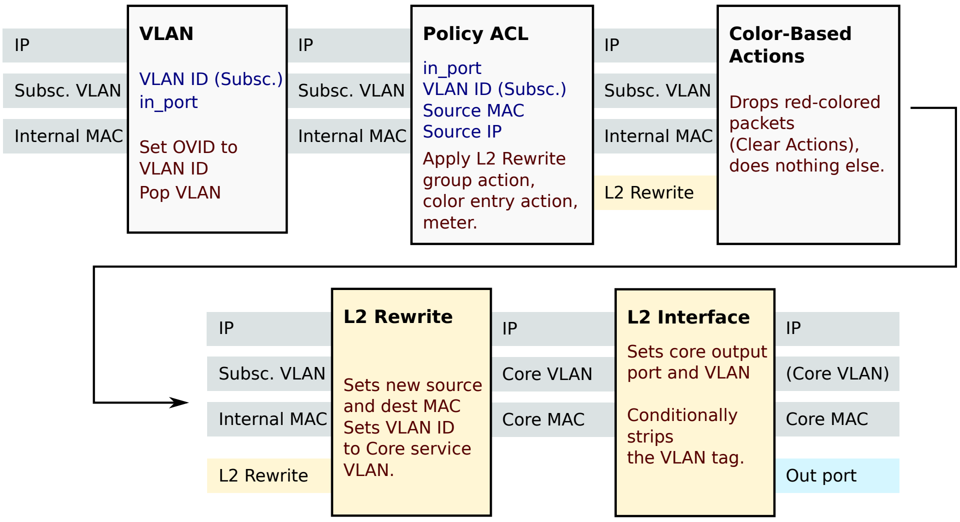


Figure 47: OF-DPA flow model of the single-tagged mode, upstream direction. [119]

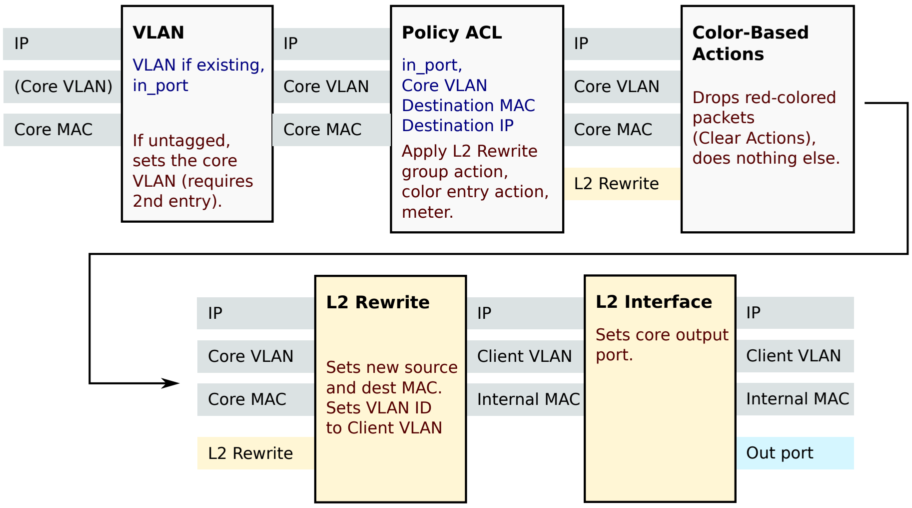


Figure 48: OF-DPA flow model of the single-tagged mode, downstream direction. [119]

four Gigabit Ethernet ports: `dslam0` (Switchport 5), `dslam1` (Switchport 6), `core0` (Switchport 7), `core1` (Switchport 8). On the test server, we have implemented two LXC containers representing CPEs (subsc1 and subsc2), every traffic from or to these containers has been tagged with an individual VLAN ID (resembling the behavior of DSLAMs) on `dslam0`. In various tests, we forwarded tagged traffic to `dslam1` instead, to verify correct behavior with multiple ports. The CPE LXC containers use a Python script as the authentication client at the BRAS.

The interface `core0` has been split into two subinterfaces for the service VLANs 8 and 10, which have been configured as tagged core interfaces in the BRAS, `core1` has been configured as the service VLAN 4, but untagged.

The connectivity between the subscriber VMs and the core networks was tested with ping and iperf3.

Evaluation results

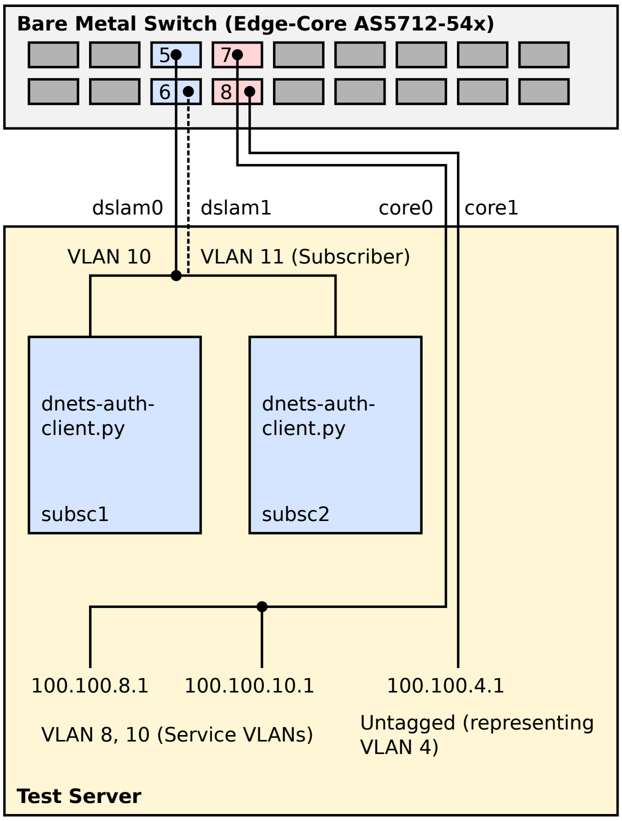


Figure 49: BRAS proof-of-concept test setup. [119]

We have focused on functional tests and table size restrictions, as we have assumed these to be the primary bottlenecks of the platform. Table 2 shows the implementation and testing status of several features of our BRAS proof of concept, we also provide whether the features are *essential* for a BRAS implementation or not: In this context, essential features are the ones that are required to build a BRAS. This e.g. applies to authentication, provisioning of leases, and ARP responding. However, VLAN double-tagging (also known as Q-in-Q or IEEE 802.1ad) can be used to increase the identifier space of VLANs, or to apply hierarchical pushing/popping of the VLANs, and is desired in carrier networks, however, it is possible to implement a functional BRAS without it.

RQ 2.3

In our prototype, we could successfully implement and test all these essential features of a BRAS. As already mentioned, we have not been able to successfully implement a functional double-tagging in the OF-DPA pipeline. At the time of writing, it was not available in the published programming interface [17]. Due to the extraordinarily high backplane capacity of the switch according to the hardware descriptions (720 gigabits per second), we could not verify an upper limit for throughput caused by the limitation of our testing equipment. Current limitations in the evaluation hardware (load generators) also did not allow to test the meters.

Scalability

Another aspect is *scalability*. We identified the Policy ACL flow table to be the primary bottleneck to create a scalable implementation on the AS5712-54x (Trident

Feature	OF-DPA Support	Impl. Status	Test Status	Essential Feature
VLAN Single Tagging	Yes	Done	Success	Yes
VLAN Double Tagging	No	-	-	No
Antispoof	Yes	Done	Success	Yes
IPv6	Yes	DP-only	Success	Yes
Metering (RFC 2697 ff.)	Yes	Done	Not tested	Yes
Authentication	-	Done	Success	Yes
Liveness	-	Done	Success	No
ARP Responder	-	Done	Success	Yes
Lease from IP Pool	-	Done	Success	Yes

Table 2: BRAS Implementation and Testing Status. [119]

II), which could manage up to 3072 entries. According to our flow model, the switch requires two flows per subscriber times subnet, in a dual-stack implementation even 4 entries are required. Even if a dual-stack subscriber uses 1 service VLAN, 768 subscribers are supported per switch. Using the Unicast Routing table for the downstream direction doubled the number of possible subscribers, however, a metering support is not intended by this table. Nevertheless, Broadcom and other vendors have announced new chipset generations with support for much larger flow tables [158]. Thus, the current scalability limits are not expected to persist.

7 CONCLUSION

In 2012, carriers started an initiative to lower the costs and to increase the flexibility of their network infrastructure, targeting to get rid of costly and inflexible appliances. By applying cloud computing principles on network functions (NFs), network functions virtualization (NFV) envisions to improve flexibility in a network to quickly enroll new kinds of services, relocate NF execution to resources more close to the subscribers, or adapt the scale of existing resources to the current demand. As motivated in the introduction, NFV is challenged by the requirements of a network to deliver more and more performance by even lower costs (resource efficiency).

Motivation (Recap)

For flexibility, it is desired that NF instances can be relocated to other resources in a network, but this must happen without any perceivable interruption of their service. Despite many efforts in research on seamless NFV *state migration* primarily over high-performance intra-datacenter (DC) links, we have shown that it is not possible to seamlessly migrate a high-performance NF over highly-utilized wide area network (WAN) links with state-of-the-art mechanisms. Providing a mechanism improving seamless instance migration over WAN links and showing its potential has been identified as our first goal.

Goal 1

To improve the performance of NFV, several efforts towards including hardware acceleration (HWA) into the NFV concept have been made. However, there is still potential to increase resource efficiency by improving elasticity between *heterogeneous* computing resources, like a commodity central processing unit (CPU) and HWA, and an immense performance potential of *merchant switch silicon* – application-specific integrated circuits (ASICs) for packet forwarding – to accelerate NFs with very low costs. Therefore, the improvement of *resource efficiency* in NFV environments using HWA has been identified as our second goal.

Goal 2

7.1 RESULTS

We have contributed the *statelet method* to announce and replay information in a packet over a well-defined interface, which is relevant for state change in an NF caused by this packet. The statelet method is intended to reduce state resynchronization traffic, which we have previously identified as a bottleneck. We have designed and implemented the *SliM* mechanism, which uses the statelet approach for NFV instance migration. Our implementation is based on state-of-the art hard-and software like the Kernel-based Virtual Machine (KVM), the Data Plane Development Kit (DPDK) and a bare-metal switch using the OpenFlow Data Plane Abstraction (OF-DPA) interface. The software is available on GitHub[1], together with the statelet interface which can be used to implement or extend other DPDK-based NFs.

Contribution 1.1: Statelet method and SliM

1 https://github.com/nokia/SliM

Contribution 1.2: Evaluation of SliM

To show the benefit of SliM, we have analyzed the mechanism on a network model with a bandwidth-restricted migration link, and compared it to packet-duplication-based deterministic replay (Duplication). We have determined that with SliM, the dataplane link utilization can almost be *tripled* during migration compared to the state of the art, because SliM only requires a fraction of the link capacity for state resynchronization traffic. Complementary to the theoretical results, a testbed evaluation under WAN-typical migration conditions confirms that SliM can successfully do a seamless migration *with almost three times as much capacity offered to the dataplane workload* as with Duplication, while being able to *finish the migration in only one third of the time*.

Contribution 2.1: An architecture for elastic HWA provisioning...

To be able to fully exploit resource utilization in infrastructures with HWA, we have proposed a split-architecture in which elastic provisioning of both commodity CPU and HWA resources is possible for different demands of simple- and complex-use-case NF workloads. After publication, parts of the architecture's idea have been also suggested by subsequent work [14, 15, 19], and parts of the proposed interfaces have been found in later standards [45, 46].

... and a model to evaluate benefits of elastic HWA provisioning

We have proposed a method to evaluate the benefits of elastic HWA provisioning. We have measured the performance and estimated the monetary costs of an NF implementation on both a field-programmable gate array (FPGA) and a commodity CPU. We have modified an elasticity benchmarking model to support *heterogeneous* processing resources, and, given a realistic daily workload model, we have determined that we can save around 40% of costs through elastic provisioning, compared to static provisioning.

Contribution 2.2: Evaluation of switch silicon functionality to implement a broadband remote access server (BRAS)

Finally, we have evaluated the functional requirements of merchant switch silicon to be able to implement a broadband remote access server (BRAS), a typical carrier-grade NF with very high performance requirements. Due to its wide availability, switch silicon can be considered as commodity hardware fitting into the NFV concept. According to the vendor's specifications, our switch platform provided a total backplane throughput of 720Gbps, which is at least *an order of magnitude higher* than same-sized commodity-CPU-based NFV platforms. Switch silicon has been originally designed for forwarding only, thus we identified scalability issues restricting the number of subscriber networks to *768*, and the lack of the platform to support the Point-to-Point Protocol (PPP) over Ethernet (PPPoE). Nevertheless, we could evaluate that all essential features of a BRAS could be implemented.

7.2 CONSEQUENCES

SliM and edge clouds

Our work *enables* migration of virtualized network function (vNF) instances in certain scenarios with WAN links between the source and the destination of a migration. This applies e.g. to future trends towards carrier *edge clouds*: NF instances placed in the proximity of a user, for example at a mobile base station or in a street cabinet, can now be seamlessly migrated, even if the uplink to a central DC is highly utilized. Other applications of SliM involve vNF instance migrations between subscriber equipment (e.g. home routers) and central DCs. Subscriber equipment is often connected via a Digital Subscriber Line (DSL) or Long-Term Evolution (LTE) link only, but vNF migration should operate without an interruption of currently active voice or video signals.

Since initial publication, the proposed split-architecture for elastic provisioning of HWA has already similarly appeared in further research and even standardization activities [14, 15, 19, 45, 46]. Our evaluation regarding the benefits of elastic provisioning can be considered as not only a motivation for further research activities, the proposed model can also be used to evaluate the benefits in actual NFV deployments, given that instance workload capacities and instance costs can be estimated. Finally, bare-metal switches can implement a BRAS NF with highest demands on performance with just one commodity switch, where commodity-CPU-based NFV would require a scale-out to multiple servers.

7.3 OUTLOOK

Based on the contributions described in this thesis, several further research and development activities are envisioned.

SliM implementation effort

We mentioned that the statelet method and SliM are not *transparent* to the NF implementation, an NF developer must adapt the NF code by implementing the statelet interface in order to support SliM. In the future, code analysis techniques, which have been similarly proposed for flow state classification [85] may be applied to identify information in a packet relevant or irrelevant for a state change at compile time, in order to exploit the advantages of the statelet interface with no or only minor development effort. Furthermore, the SliM implementation could be extended with partial migration as already described in the SliM design, in order to support split and merge functionality like in OpenNF [57]. In rare cases, delta-based virtual machine (VM) migration might perform better than SliM *during the first delta rounds*, as invalidated state in a packet ring buffer becomes overwritten after some time. It might be therefore interesting to analyze how performance could be improved using *hybrid* state migration mechanisms, which first transfer several rounds of state deltas before starting to announce statelets.

SliM partial migration

Using SliM for HWA provisioning

Finally, SliM might be applied as a solution to the state migration problem of elastic hardware accelerator provisioning. Due to the effort of implementing SliM on hardware accelerators we could not finish a contribution to *combine* the two problem areas of our dissertation. Regarding the implementation of NFs on bare-metal switching ASICs, there is an ongoing effort in industry towards switch silicon fully programmable with description languages like Programming Protocol-Independent Packet Processors (P4) [11], for example the *Barefoot Tofino* or the *Cavium XPliant* chipset. The upcoming silicon generations will likely support functionality which could not be implemented on our switch platform.

BIBLIOGRAPHY

[1] R. W. Ahmad, A. Gani, S. H. A. Hamid, M. Shiraz, A. Yousafzai, and F. Xia, "A survey on virtual machine migration and server consolidation frameworks for cloud data centers," *Journal of Network and Computer Applications (JNCA)*, vol. 52, pp. 11–25, 2015 (cit. on pp. 28, 43, 50, 54, 56, 65).

[2] S. Aketa, T. Hirofuchi, and R. Takano, "Demu: A dpdk-based network latency emulator," in *International Symposium on Local and Metropolitan Area Networks (LANMAN)*, IEEE, Jun. 2017, pp. 1–6 (cit. on p. 69).

[3] B. Anwer, T. Benson, N. Feamster, and D. Levin, "Programming slick network functions," in *Symposium on SDN Research (SOSR)*, ACM, 2015, p. 14 (cit. on p. 30).

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *SIGOPS operating systems review (OSR)*, ACM, vol. 37, 2003, pp. 164–177 (cit. on p. 7).

[5] A. Basta, A. Blenk, K. Hoffmann, H. J. Morper, M. Hoffmann, and W. Kellerer, "Towards a cost optimal design for a 5G mobile core network based on SDN and NFV," *Transactions on Network and Service Management (TNSM)*, vol. 14, no. 4, pp. 1061–1075, Dec. 2017 (cit. on p. 12).

[6] A. Basta, W. Kellerer, M. Hoffmann, H. J. Morper, and K. Hoffmann, "Applying NFV and SDN to LTE mobile core gateways, the functions placement problem," in *SIGCOMM AllThingsCellular Workshop*, ACM, 2014, pp. 33–38 (cit. on p. 12).

[7] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The turtles project: Design and implementation of nested virtualization," in *Symposium on Operating Systems Design and Implementation (OSDI)*, USENIX, 2010 (cit. on p. 22).

[8] R. Bifulco, T. Dietz, F. Huici, M. Ahmed, J. Martins, S. Niccolini, and H.-J. Kolbe, "Rethinking access networks with high performance virtual software BRASes," in *European Workshop on Software-Defined Networks (EWSDN)*, IEEE, 2013 (cit. on p. 12).

[9] J. Blendin, J. Rückert, N. Leymann, G. Schyguda, and D. Hausheer, "Position paper: Software-defined network service chaining," in *European Workshop on Software-Defined Networks (EWSDN)*, IEEE, 2014 (cit. on pp. 10, 13).

[10] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Workshop on Mobile Cloud Computing (MCC)*, ACM, 2012, pp. 13–16 (cit. on p. 2).

[11] 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," *SIGCOMM Computer Communication Review (CCR)*, vol. 44, no. 3, pp. 87–95, 2014 (cit. on pp. 24, 76, 81, 109).

[12] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, Oct. 1977 (cit. on p. 32).

[13] G. Brebner, "Softly Defined Networking," in *Symposium on Architectures for Networking and Communications Systems (ANCS)*, ACM, 2012, pp. 1–2 (cit. on pp. 3, 31).

[14] A. Bremler-Barr, Y. Harchol, and D. Hay, "OpenBox: Enabling innovation in middlebox applications," in *SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*, ACM, 2015 (cit. on pp. 4, 32, 37, 108, 109).

[15] A. Bremler-Barr, Y. Harchol, and D. Hay, "OpenBox: A software-defined framework for developing, deploying, and managing network functions," in *SIGCOMM*, ACM, 2016, pp. 511–524 (cit. on pp. 32, 108, 109).

[16] Broadcom, *Broadcom-Switch on GitHub*, Dec. 2016. [Online]. Available: https://github.com/Broadcom-Switch/ (cit. on p. 97).

[17] Broadcom, *OpenFlow Data Plane Abstraction (OF-DPA): Abstract Switch Specification, Version 2.01*, Dec. 2016. [Online]. Available: https://github.com/Broadcom-Switch/of-dpa/blob/master/OFDPAS-ETP100-R.pdf (cit. on pp. 11, 14, 25, 95, 97, 98, 105).

[18] Broadcom, *OpenFlow Data Plane Abstraction (OF-DPA): Python wrapper*, Dec. 2016. [Online]. Available: https://github.com/Broadcom-Switch/of-dpa/blob/master/bin/as6700-trident2-fsl14/OFDPA_python.py (cit. on p. 97).

[19] 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, May 2015 (cit. on pp. 32, 108, 109).

[20] S. Byma, J. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow, "FPGAs in the cloud: Booting virtualized hardware accelerators with OpenStack," in *Field-Programmable Custom Computing Machines (FCCM)*, IEEE, May 2014, pp. 109–116 (cit. on pp. 3, 23, 31, 33, 75).

[21] G. Carella, M. Corici, P. Crosta, P. Comi, T. Bohnert, A. Corici, D. Vingarzan, and T. Magedanz, "Cloudified IP multimedia subsystem (IMS) for network function virtualization (NFV)-based architectures," in *International Symposium on Computers and Communication (ISCC)*, IEEE, 2014 (cit. on p. 12).

[22] B. Carpenter and S. Brim, "Middleboxes: Taxonomy and issues," RFC 3234, Feb. 2002 (cit. on pp. 8, 9).

[23] I. Cerrato, M. Annarumma, and F. Risso, "Supporting fine-grained network functions through Intel DPDK," in *European Workshop on Software-Defined Networks (EWSDN)*, IEEE, 2014 (cit. on pp. 21, 35).

[24] N. M. K. Chowdhury and R. Boutaba, "A survey of network virtualization," *Elsevier Computer Networks*, vol. 54, no. 5, pp. 862–876, 2010 (cit. on p. 13).

[25] Cisco Systems, "Cisco Global Cloud Index: Forecast and Methodology, 2016–2021 White Paper," Tech. Rep., May 2018. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.pdf (cit. on pp. 1, 2).

[26] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Networked Systems Design and Implementation (NSDI)*, USENIX, 2005, pp. 273–286 (cit. on pp. 2, 27, 30, 35, 37, 39, 40, 43, 44, 50, 54, 65).

[27] CORD Project, *About OpenCORD*. [Online]. Available: https://opencord.org/about/ (cit. on pp. 11, 33).

[28] P. Cortez, M. Rio, M. Rocha, and P. Sousa, "Multi-scale internet traffic forecasting using neural networks and time series methods," *Expert Systems*, vol. 29, no. 2, pp. 143–155, 2012 (cit. on pp. 5, 84, 85, 94).

[29] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *Networked Systems Design and Implementation (NSDI)*, USENIX, 2008, pp. 161–174 (cit. on p. 27).

[30] S. Davy, J. Famaey, J. Serrat-Fernandez, J. Gorricho, A. Miron, M. Dramitinos, P. Neves, S. Latre, and E. Goshen, "Challenges to support edge-as-a-service," *IEEE Communications Magazine*, vol. 52, no. 1, pp. 132–139, Jan. 2014 (cit. on p. 12).

[31] S. W. Devine, E. Bugnion, and M. Rosenblum, *Virtualization system including a virtual machine monitor for a computer with a segmented architecture*, US Patent 6,397,242, May 2002 (cit. on p. 7).

[32] T. Dietz, R. Bifulco, F. Manco, J. Martins, H.-J. Kolbe, and F. Huici, "Enhancing the BRAS through virtualization," in *Conference on Network Softwarization (NetSoft)*, IEEE, 2015, pp. 1–5 (cit. on p. 28).

[33] M. Dillon and T. Winters, "Virtualization of home network gateways," *IEEE Computer*, vol. 47, no. 11, pp. 62–65, Nov. 2014 (cit. on p. 12).

[34] Y. Dong, D. Xu, Y. Zhang, and G. Liao, "Optimizing network I/O virtualization with efficient interrupt coalescing and virtual receive side scaling," in *International Conference on Cluster Computing (CLUSTER)*, IEEE, 2011 (cit. on p. 22).

[35] DPDK Project, *DPDK API documentation*. [Online]. Available: http://dpdk.org/doc/api/ (cit. on pp. 22, 35, 56).

[36] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "Revirt: Enabling intrusion analysis through virtual-machine logging and replay," *SIGOPS Operating Systems Review (OSR)*, vol. 36, no. SI, pp. 211–224, 2002 (cit. on p. 29).

[37] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A scriptable high-speed packet generator," in *Internet Measurement Conference (IMC)*, ACM, Tokyo, Japan, Oct. 2015 (cit. on pp. 82, 131).

[38] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, "Performance characteristics of virtual switching," in *International Conference on Cloud Networking (CloudNet)*, IEEE, 2014, pp. 120–125 (cit. on p. 20).

[39] European Telecommunication Standards Institute (ETSI), *ETSI NFV architecture with the relevant documents*. [Online]. Available: http://www.etsi.org/images/articles/NFV%5C%20Architecture.svg (cit. on p. 15).

[40] European Telecommunication Standards Institute (ETSI), *ETSI NFV open documents*. [Online]. Available: https://docbox.etsi.org/isg/nfv/open (cit. on p. 10).

[41] European Telecommunication Standards Institute (ETSI), *ETSI NFV release 3 definition (accessed 2018-05)*. [Online]. Available: https://docbox.etsi.org/isg/nfv/open/Other/NFV(16)000229r11_NFV_Release_3_Definition_v0_10_0.pdf (cit. on p. 10).

[42] European Telecommunication Standards Institute (ETSI), *LTE; general packet radio service (GPRS) enhancements for evolved universal terrestrial radio access network (E-UTRAN) access*, Technical Specification, 3GPP TS 23.401 version 8.14.0 Release 8, Jun. 2011 (cit. on p. 42).

[43] European Telecommunication Standards Institute (ETSI), *ETSI: Network functions virtualisation (NFV); use cases*, Group Specification, 2013 (cit. on p. 11).

[44] European Telecommunication Standards Institute (ETSI), *ETSI GS NFV 003: Network functions virtualisation (NFV); terminology for main concepts in NFV v1.2.1*, Group Specification, Dec. 2014 (cit. on pp. 8, 10).

[45] European Telecommunication Standards Institute (ETSI), *ETSI GS NFV-IFA 019: Network functions virtualisation (NFV); acceleration technologies; acceleration resource management interface specification; release 3*, Group Specification, 2017 (cit. on pp. 32, 108, 109).

[46] European Telecommunication Standards Institute (ETSI), *ETSI GS NFV-IFA 004: Network functions virtualisation (NFV) release 2; acceleration technologies; management aspects specification*, Group Specification, 2018 (cit. on pp. 32, 108, 109).

[47] European Telecommunication Standards Institute (ETSI), *ETSI GS NFV-IFA 008: Network functions virtualisation (NFV) release 2; management and orchestration; Ve-Vnfm reference point - interface and information model specification*, Group Specification, 2018 (cit. on p. 14).

[48] Facebook, "Facebook open switching system (FBOSS)," Tech. Rep., Nov. 2013. [Online]. Available: https://code.facebook.com/posts/681382905244727 (cit. on p. 24).

[49] S. A. Fahmy, K. Vipin, and S. Shreejith, "Virtualized fpga accelerators for efficient cloud computing," in *International Conference on Cloud Computing Technology and Science (CloudCom)*, IEEE, Nov. 2015, pp. 430–435 (cit. on p. 32).

[50] S. K. Fayaz and V. Sekar, "Testing stateful and dynamic data planes with FlowTest," in *SIGCOMM Workshop on Hot topics in software defined networking (HotSDN)*, ACM, 2014, pp. 79–84 (cit. on p. 30).

[51] N. Figueira and R. Krishnan, *Policy architecture and framework for NFV and cloud services*, IETF Draft, ID: draft-norival-nfvrg-nfv-policy-arch-01, Feb. 2015 (cit. on p. 11).

[52] A. G. Forte, W. Wang, L. Veltri, and G. Ferrari, "A P2P virtual core-network architecture for next-generation mobility networks," in *Standards for Communications and Networking (CSCN)*, IEEE, 2016, pp. 1–7 (cit. on p. 16).

[53] D. Fritzsche, Z. Magyari, M. Schlosser, and T. Jungel, "Basebox – integrating whitebox switches into linux: A controller implementation for OF-DPA hardware," in *European Workshop on Software-Defined Networks (EWSDN)*, IEEE, 2016, pp. 52–54 (cit. on pp. 33, 95).

[54] J. García-Dorado, F. Mata, J. Ramos, P. Santiago del Río, V. Moreno, and J. Aracil, "High-performance network traffic processing systems using commodity hardware," in *Data Traffic Monitoring and Analysis*, vol. 7754, Springer Berlin Heidelberg, 2013, pp. 3–27, ISBN: 978-3-642-36783-0 (cit. on pp. 19, 22).

[55] X. Ge, Y. Liu, D. H. Du, L. Zhang, H. Guan, J. Chen, Y. Zhao, and X. Hu, "OpenANFV: Accelerating network function virtualization with a consolidated framework in OpenStack," in *SIGCOMM*, ACM, 2014, pp. 353–354 (cit. on pp. 3, 31, 75).

[56] A. Gember-Jacobson and A. Akella, "Improving the safety, scalability, and efficiency of network function state transfers," in *SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*, ACM, 2015 (cit. on pp. 4, 29, 30, 35, 37, 39, 40, 44, 50, 65).

[57] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "OpenNF: Enabling innovation in network function control," in *SIGCOMM*, ACM, 2014 (cit. on pp. 2, 4, 17, 29, 30, 35, 37, 39, 40, 43, 44, 49, 65, 109).

[58] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella, "Toward software-defined middlebox networking," in *Workshop on Hot Topics in Networks (HotNets)*, ACM, 2012, pp. 7–12 (cit. on p. 29).

[59] G. A. Gibson and R. Van Meter, "Network attached storage architecture," *Communications of the ACM*, vol. 43, no. 11, pp. 37–45, 2000 (cit. on p. 75).

[60] A. Gordon, N. Har'El, A. Landau, M. Ben-Yehuda, and A. Traeger, "Towards exitless and efficient paravirtual I/O," in *SIGOPS Systems and Storage Conference (SYSTOR)*, ACM, 2012 (cit. on p. 22).

[61] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-accelerated software router," in *SIGCOMM*, ACM, 2010 (cit. on p. 31).

[62] R. Hans, D. Steffen, U. Lampe, B. Richerzhagen, and R. Steinmetz, "Setting priorities-a heuristic approach for cloud data center selection," in *International Conference on Cloud Computing and Services Science (CLOSER)*, 2015, pp. 221–228 (cit. on p. 2).

[63] J. Hautakorpi, G. Camarillo, R. Penfield, A. Hawrylyshen, and M. Bhatia, "Requirements from session initiation protocol (SIP) session border control (SBC) deployments," RFC 5853, Apr. 2010 (cit. on pp. 12, 77).

[64] D. Heldenbrand and C. Carey, "The linux router: An inexpensive alternative to commercial routers in the lab," *Journal of Computing Sciences in Colleges (JCSC)*, vol. 23, no. 1, pp. 127–133, Oct. 2007 (cit. on p. 10).

[65] S. Hemminger *et al.*, "Network emulation with netem," in *Australasia's Regional Linux and Open Source Conference (LCA)*, 2005, pp. 18–23. [Online]. Available: https://www.rationali.st/blog/files/20151126-jittertrap/netem-shemminger.pdf (cit. on p. 68).

[66] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in cloud computing: What it is, and what it is not," in *International Conference on Autonomic Computing (ICAC)*, USENIX, 2013, pp. 23–27 (cit. on pp. 16, 32, 75, 83, 94).

[67] N. Herbst, R. Krebs, G. Oikonomou, G. Kousiouris, A. Evangelinou, A. Iosup, and S. Kounev, "Ready for rain? a view from SPEC research on the future of cloud metrics," Standard Performance Evaluation Corporation (SPEC) Research Group, Technical Report SPEC-RG-2016-01, Mar. 2016. [Online]. Available: https://research.spec.org/fileadmin/user_upload/documents/rg_cloud/endorsed_publications/SPEC-RG-2016-01_CloudMetrics.pdf (cit. on pp. 16, 32, 75, 83, 84, 91, 94).

[68] J. G. Herrera and J. F. Botero, "Resource allocation in NFV: A comprehensive survey," *Transactions on Network and Service Management (TNSM)*, vol. 13, no. 3, pp. 518–532, Sep. 2016 (cit. on p. 2).

[69] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, and B. Koldehofe, "Mobile fog: A programming model for large-scale applications on the internet of things," in *SIGCOMM Workshop on Mobile Cloud Computing (MCC)*, ACM, Hong Kong, China, 2013, pp. 15–20 (cit. on p. 2).

[70] Z. Huang, R. Ma, J. Li, Z. Chang, and H. Guan, "Adaptive and scalable optimizations for high performance SR-IOV," in *International Conference on Cluster Computing (CLUSTER)*, IEEE, 2012 (cit. on p. 22).

[71] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: High performance and flexible networking using virtualization on commodity platforms," in *Networked Systems Design and Implementation (NSDI)*, USENIX, 2014 (cit. on p. 19).

[72] C.-L. I, J. Huang, R. Duan, C. Cui, J. Jiang, and L. Li, "Recent progress on C-RAN centralization and cloudification," *Access*, vol. 2, pp. 1030–1039, 2014 (cit. on p. 12).

[73] *IETF working group for service function chaining (SFC)*. [Online]. Available: https://datatracker.ietf.org/wg/sfc/documents/ (cit. on p. 11).

[74] Intel, "Network function virtualization: Packet processing peformance of virtualized platforms with Linux* and Intel® architecture," Tech. Rep., Oct. 2013 (cit. on p. 23).

[75] International Telecommunication Union (ITU), "ITU-T Q.1002 - switching and signalling," Recommendation, Nov. 1988, superseded on 24/12/2003. [Online]. Available: http://handle.itu.int/11.1002/1000/1779 (cit. on p. 1).

[76] International Telecommunication Union (ITU), "ITU-T G.1080 - quality of experience requirements for IPTV services," Recommendation, Dec. 2008. [Online]. Available: https://www.itu.int/rec/T-REC-G.1080-200812-I/en (cit. on p. 18).

[77] International Telecommunication Union (ITU), "ITU-T P.1010 - fundamental voice transmission objectives for VoIP terminals and gateways," Recommendation, Dec. 2008. [Online]. Available: https://www.itu.int/rec/T-REC-P.1010-200407-I/en (cit. on p. 18).

[78] International Telecommunication Union (ITU), "ITU-T Y.1541 - network performance objectives for IP-based services," Recommendation, Dec. 2011. [Online]. Available: https://www.itu.int/rec/T-REC-Y.1541-201112-I/en (cit. on p. 18).

[79] Internet Engineering Task Force (IETF), *Network function virtualization research group (NFVRG)*. [Online]. Available: https://trac.tools.ietf.org/group/irtf/trac/wiki/nfvrg (cit. on pp. 10, 11).

[80] Internet Systems Consortium, *Address family transition router (AFTR)*, Dec. 2014. [Online]. Available: http://www.isc.org/downloads/aftr/ (cit. on p. 12).

[81] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: A highly scalable user-level TCP stack for multicore systems," in *Networked Systems Design and Implementation (NSDI)*, USENIX, 2014, pp. 489–502 (cit. on p. 58).

[82] M. Kablan, B. Caldwell, R. Han, H. Jamjoom, and E. Keller, "Stateless network functions," in *SIGCOMM Workshop on Hot topics in software defined networking (HotSDN)*, ACM, 2015, pp. 49–54 (cit. on p. 30).

[83] C. Kachris, G. Sirakoulis, and D. Soudris, “Network Function Virtualization based on FPGAs: A Framework for All-Programmable Network Devices,” *arXiv:1406.0309*, 2014 (cit. on pp. 3, 23, 31).

[84] K. Karras and J. Hrica, “Designing protocol processing systems with vivado high-level synthesis,” Xilinx, Tech. Rep. XAPP1209, Aug. 2014. [Online]. Available: `http://www.xilinx.com/support/documentation/application_notes/xapp1209-designing-protocol-processing-systems-hls.pdf` (cit. on p. 130).

[85] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella, “Paving the way for NFV: Simplifying middlebox modifications using StateAlyzr,” in *Networked Systems Design and Implementation (NSDI)*, USENIX, 2016, pp. 239–253 (cit. on pp. 43, 109).

[86] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, “The click modular router,” *Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000 (cit. on p. 10).

[87] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-defined networking: A comprehensive survey,” *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015 (cit. on p. 13).

[88] R. Krishnan, N. Figueira, D. Krishnaswamy, D. R. Lopez, S. Wright, and T. Hinrichs, *NFVIaaS architectural framework for policy based resource placement and scheduling*, IETF Draft, ID: draft-krishnan-nfvrg-policy-based-rm-nfviaas-03, Nov. 2014 (cit. on p. 11).

[89] R. Krishnan, D. Krishnaswamy, D. R. Lopez, A. Qamar, S. Wright, and N. Figueira, *NFV real-time analytics and orchestration: Use cases and architectural framework*, IETF Draft, ID: draft-krishnan-nfvrg-real-time-analytics-orch-01, Nov. 2014 (cit. on p. 11).

[90] R. Krishnan, D. Krishnaswamy, and D. Mcdysan, “Behavioral security threat detection strategies for data center switches and routers,” in *International Conference on Distributed Computing Systems (ICDCS)*, IEEE, Jun. 2014, pp. 82–87 (cit. on p. 13).

[91] S. Kuenzer, J. Martins, M. Ahmed, and F. Huici, “Towards minimalistic, virtualized content caches with minicache,” in *SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*, ACM, Santa Barbara, California, USA, 2013 (cit. on p. 13).

[92] F. Kurtz, N. Dorsch, and C. Wietfeld, “Empirical comparison of virtualized and bare-metal switching for SDN-based 5G communication in critical infrastructures,” in *Conference on Network Softwarization (NetSoft)*, Jun. 2016, pp. 453–458 (cit. on pp. 33, 95).

[93] S. Lee, S. Pack, M.-K. Shin, and E. Paik, *Resource management for dynamic service chain adaptation*, IETF Draft, ID: draft-lee-nfvrg-resource-management-service-chain-00, 2014 (cit. on p. 11).

[94] P. Lieser, N. Richerzhagen, T. Feuerbach, L. Nobach, D. Böhnstedt, and R. Steinmetz, "Take it or leave it: Decentralized resource allocation in mobile networks," in *Conference on Local Computer Networks (LCN)*, IEEE, 2017.

[95] Y.-D. Lin, "Research roadmap driven by network benchmarking lab (NBL): Deep packet inspection, traffic forensics, embedded benchmarking, software defined networking and beyond," *International Journal of Networking and Computing (IJNC)*, vol. 4, no. 2, pp. 223–235, 2014 (cit. on p. 13).

[96] Linux Foundation, *New API (NAPI)*. [Online]. Available: `https://wiki.linuxfoundation.org/networking/napi` (cit. on p. 22).

[97] H. Liu, H. Jin, X. Liao, L. Hu, and C. Yu, "Live migration of virtual machine based on full system trace and replay," in *High-Performance Parallel and Distributed Computing (HPDC)*, ACM, 2009, pp. 101–110 (cit. on pp. 2, 29, 44).

[98] J. Liu, H. Shen, and H. Hu, "Load-aware and congestion-free state management in network function virtualization," in *Computing, Networking and Communications (ICNC)*, IEEE, 2017, pp. 303–307 (cit. on p. 29).

[99] J. Liu, "Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support," in *International Parallel & Distributed Processing Symposium (IPDPS)*, IEEE, 2010 (cit. on p. 21).

[100] L. Liu, H. Xu, Z. Niu, P. Wang, and D. Han, "U-HAUL: Efficient state migration in NFV," in *SIGOPS Asia-Pacific Workshop on Systems*, ACM, 2016, p. 2 (cit. on p. 30).

[101] Y. Liu, W. Gong, and P. Shenoy, "On the impact of concurrent downloads," in *Winter Simulation Conference (WSC)*, vol. 2, 2001, 1300–1305 vol.2 (cit. on p. 64).

[102] A. Lometti, C. Addeo, I. Busi, and V. Sestito, "Backhauling solutions for lte networks," in *International Conference on Transparent Optical Networks (ICTON)*, IEEE, 2014, pp. 1–6 (cit. on p. 12).

[103] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. J. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam, *et al.*, "Jitsu: Just-in-time summoning of unikernels.," in *Networked Systems Design and Implementation (NSDI)*, USENIX, 2015, pp. 559–573 (cit. on p. 23).

[104] L. Mamakos, K. Lidl, J. Evarts, D. Carrel, D. Simone, and R. Wheeler, "A method for transmitting PPP over ethernet (PPPoE)," RFC 2516, Feb. 1999 (cit. on p. 76).

[105] I. Marinos, R. N. Watson, and M. Handley, "Network stack specialization for performance," in *SIGCOMM*, ACM, vol. 44, 2014, pp. 175–186 (cit. on pp. 19, 21, 95).

[106] J. Martins, M. Ahmed, C. Raiciu, and F. Huici, "Enabling fast, dynamic network processing with ClickOS," in *SIGCOMM Workshop on Hot topics in software defined networking (HotSDN)*, ACM, Hong Kong, China, 2013, pp. 67–72 (cit. on pp. 19, 23, 28).

[107] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "ClickOS and the art of network function virtualization," in *Networked Systems Design and Implementation (NSDI)*, USENIX, 2014 (cit. on pp. 19, 23, 28).

[108] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Computer Communication Review (CCR)*, vol. 38, no. 2, pp. 69–74, 2008 (cit. on pp. 33, 75).

[109] C. Meirosu, A. Manzalini, J. Kim, R. Steinert, S. Sharma, and G. Marchetto, *DevOps for software-defined telecom infrastructures*, IETF Draft, ID: draft-unify-nfvrg-devops-00, Oct. 2014 (cit. on p. 11).

[110] R. A. Meyer and L. H. Seawright, "A virtual machine time-sharing system," *IBM Systems Journal*, vol. 9, no. 3, pp. 199–218, 1970 (cit. on p. 7).

[111] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2016 (cit. on p. 10).

[112] G. Monteleone and P. Paglierani, "Session border controller virtualization towards "service-defined"networks based on NFV and SDN," in *IEEE Software Defined Networks for Future Networks and Services (SDN4FNS)*, IEEE, Nov. 2013, pp. 1–7 (cit. on pp. 12, 77).

[113] D. Murray and T. Koziniec, "The state of enterprise network traffic in 2012," in *Asia-Pacific Conference on Communications (APCC)*, IEEE, Oct. 2012, pp. 179–184 (cit. on p. 88).

[114] "Network functions virtualisation - introductory white paper," Tech. Rep., 2012. [Online]. Available: https://portal.etsi.org/NFV/NFV_White_Paper.pdf (cit. on pp. 1, 10).

[115] V. G. Nguyen and Y. H. Kim, "Slicing the next mobile packet core network," in *International Symposium on Wireless Communication Systems (ISWCS)*, IEEE, 2014, pp. 901–904 (cit. on p. 12).

[116] N. Nikaein, M. K. Marina, S. Manickam, A. Dawson, R. Knopp, and C. Bonnet, "OpenAirInterface: A flexible platform for 5G research," *SIGCOMM Computer Communication Review (CCR)*, vol. 44, no. 5, pp. 33–38, Oct. 2014 (cit. on p. 12).

[117] H. Niu, C. Li, A. Papathanassiou, and G. Wu, "RAN architecture options and performance for 5G network evolution," in *Wireless Communications and Networking Conference Workshops (WCNCW)*, IEEE, 2014, pp. 294–298 (cit. on p. 12).

[118] L. Nobach, B. Rudolph, and D. Hausheer, "Benefits of conditional FPGA provisioning for virtualized network functions," in *International Conference and Workshops on Networked Systems (NetSys)*, IEEE, Mar. 2017, pp. 1–6 (cit. on pp. 75, 82, 86, 87, 129, 145).

[119] L. Nobach, J. Blendin, H.-J. Kolbe, G. Schyguda, and D. Hausheer, "Bare-metal switches and their customization and usability in a carrier-grade environment," English, in *Conference on Local Computer Networks (LCN)*, IEEE, Oct. 2017, pp. 649–657 (cit. on pp. 7, 95, 96, 100–106, 132, 134–136, 140, 141, 145).

[120] L. Nobach, J. Blendin, H.-J. Kolbe, G. Schyguda, and D. Hausheer, "RTP packet loss healing on a bare-metal switch (demo)," in *Network Operations and Management Symposium (NOMS)*, Apr. 2018.

[121] L. Nobach and D. Hausheer, "Towards decentralized, energy- and privacy-aware device-to-device content delivery," in *Monitoring and Securing Virtualized Networks and Services*, Springer, 2014, pp. 128–132.

[122] L. Nobach and D. Hausheer, "Open, elastic provisioning of hardware acceleration in NFV environments," in *International Conference and Workshops on Networked Systems (NetSys)*, IEEE, Mar. 2015, pp. 1–5 (cit. on pp. 15, 31, 75, 79, 80, 145).

[123] L. Nobach and D. Hausheer, "PrivateShare: Measuring device-to-device user behavior and transmission quality," in *Network Operations and Management Symposium (NOMS)*, IEEE/IFIP, Apr. 2016, pp. 830–833.

[124] L. Nobach, O. Hohlfeld, and D. Hausheer, "New kid on the block: Network functions virtualization: From big boxes to carrier clouds (editorial)," *SIGCOMM Computer Communication Review (CCR)*, Jul. 2016 (cit. on pp. 7, 20).

[125] L. Nobach, Y. L. Louédec, and D. Hausheer, "Evaluating device-to-device content delivery potential on a mobile ISP's dataset," in *International Conference on Network and Service Management (CNSM)*, Nov. 2015, pp. 301–309.

[126] L. Nobach, I. Rimac, V. Hilt, and D. Hausheer, "SliM: Enabling efficient, seamless NFV state migration," in *International Conference on Network Protocols (ICNP)*, IEEE, Nov. 2016, pp. 1–2 (cit. on p. 35).

[127] L. Nobach, I. Rimac, V. Hilt, and D. Hausheer, "Statelet-based efficient and seamless NFV state transfer," English, *Transactions on Network and Service Management (TNSM)*, vol. 14, no. 4, pp. 964–977, Dec. 2017 (cit. on pp. 35, 39, 45, 46, 48, 51, 54, 55, 57, 65, 66, 70–74).

[128] V. Olteanu, F. Huici, and C. Raiciu, "Lost in network address translation: Lessons from scaling the world's simplest middlebox," in *SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*, ACM, 2015 (cit. on pp. 8, 28).

[129] L. Peterson, A. Al-Shabibi, T. Anshutz, S. Baker, A. Bavier, S. Das, J. Hart, G. Palukar, and W. Snow, "Central office re-architected as a data center," *Communications Magazine*, vol. 54, no. 10, pp. 96–101, 2016 (cit. on pp. 11, 33).

[130] J. Pettit, J. Gross, B. Pfaff, M. Casado, and S. Crosby, "Virtual switching in an era of advanced edges," in *Workshop on Data Center–Converged and Virtual Ethernet Switching (DC-CAVES)*, 2010 (cit. on p. 19).

[131] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker, "Extending networking into the virtualization layer.," in *Workshop on Hot Topics in Networks (HotNets)*, ACM, 2009 (cit. on p. 19).

[132] Pica8, *PicOS datasheet*, Jun. 2016. [Online]. Available: https://www.pica8.com/wp-content/uploads/Pica8-Datasheet-1.pdf (cit. on pp. 25, 97).

[133] G. Pongrácz, L. Molnár, Z. L. Kis, and Z. Turányi, "Cheap Silicon: A Myth or Reality? Picking the Right Data Plane Hardware for Software Defined Networking," in *SIGCOMM Workshop on Hot topics in software defined networking (HotSDN)*, ACM, 2013, pp. 103–108 (cit. on p. 75).

[134] I. Pratt and K. Fraser, "Arsenic: A User-Accessible Gigabit Ethernet Interface," in *International Conference on Computer Communications (INFOCOM)*, IEEE, 2001 (cit. on p. 22).

[135] QEmu Project, *IVSHMEM implementation*. [Online]. Available: https://github.com/qemu/qemu/blob/master/hw/misc/ivshmem.c (cit. on p. 20).

[136] Z. Qiang, *Elasticity VNF*, IETF Draft, ID: draft-zu-nfvrg-elasticity-vnf-00, Oct. 2014 (cit. on p. 11).

[137] P. Raad, S. Secci, D. C. Phung, A. Cianfrani, P. Gallard, and G. Pujolle, "Achieving sub-second downtimes in large-scale virtual machine migrations with LISP," *Transactions on Network and Service Management (TNSM)*, vol. 11, no. 2, pp. 133–143, 2014 (cit. on p. 28).

[138] S. Rajagopalan, "System support for elasticity and high availability," PhD thesis, University of British Columbia, 2014 (cit. on p. 27).

[139] S. Rajagopalan, D. Williams, and H. Jamjoom, "Pico replication: A high availability framework for middleboxes," in *Symposium on Cloud Computing (SoCC)*, ACM, 2013 (cit. on p. 27).

[140] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/Merge: System support for elastic execution in virtual middleboxes," in *Networked Systems Design and Implementation (NSDI)*, USENIX, 2013 (cit. on pp. 2, 4, 16, 17, 28, 29, 31, 41, 49).

[141] M. Rash, *Linux Firewalls: Attack Detection and Response with iptables, psad, and fwsnort*. No Starch Press, 2007 (cit. on p. 10).

[142] I. Rimac, L. Nobach, and V. Hilt, *Method for migration of virtual network function*, EP Patent App. EP20,160,290,053, Sep. 2017.

[143] L. Rizzo and G. Lettieri, "VALE, a switched ethernet for virtual machines," in *Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2012 (cit. on p. 20).

[144] B. Rudolph, "Evaluating benefits of elastic FPGA accelerator provisioning in NFV environments," Master's thesis, TU Darmstadt, 2016 (cit. on pp. 82, 86, 87, 129).

[145] F. Schmidt, O. Hohlfeld, R. Glebke, and K. Wehrle, "Santa: Faster packet delivery for commonly wished replies," in *SIGCOMM Computer Communication Review (CCR)*, ACM, vol. 45, 2015, pp. 597–598 (cit. on pp. 19, 22).

[146] L. H. Seawright and R. A. MacKinnon, "VM/370—a study of multiplicity and usefulness," *IBM Systems Journal*, vol. 18, no. 1, pp. 4–17, 1979 (cit. on p. 7).

[147] J. Sherry, P. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, *et al.*, "Rollback recovery for middleboxes," in *SIGCOMM*, ACM, 2015 (cit. on p. 27).

[148] M.-K. Shin, K. Nam, S. Pack, and S. Lee, *Verification of NFV services: Problem statement and architecture*, IETF Draft, 2014 (cit. on p. 11).

[149] V. Sivaraman, A. Vishwanath, Z. Zhao, and C. Russell, "Profiling per-packet and per-byte energy consumption in the NetFPGA gigabit router," in *INFOCOM Computer Communications Workshops (WKSHPS)*, IEEE, Apr. 2011, pp. 331–336 (cit. on p. 88).

[150] R. Steinmetz, *Multimedia-Technologie: Grundlagen, Komponenten und Systeme*, 3rd ed. Springer-Verlag, 2013 (cit. on p. 18).

[151] R. Steinmetz and K. Wehrle, "What is this "peer-to-peer" about?" In *Peer-to-peer systems and applications*, Springer, 2005, pp. 9–16 (cit. on p. 16).

[152] D. Stingl, C. Gross, L. Nobach, R. Steinmetz, and D. Hausheer, "BlockTree: Location-aware decentralized monitoring in mobile ad hoc networks," in *Conference on Local Computer Networks (LCN)*, IEEE, 2013, pp. 373–381.

[153] D. Stingl, C. Gross, J. Rückert, L. Nobach, A. Kovacevic, and R. Steinmetz, "PeerfactSim.KOM: A simulation framework for peer-to-peer systems," in *Conference on High Performance Computing and Simulation (HPCS)*, IEEE, 2011, pp. 577–584.

[154] P. Svärd, B. Hudzia, J. Tordsson, and E. Elmroth, "Evaluation of delta compression techniques for efficient live migration of large virtual machines," *Sigplan Notices*, vol. 46, no. 7, pp. 111–120, 2011 (cit. on pp. 28, 65).

[155] R. Szabo, A. Csaszar, K. Pentikousis, M. Kind, and D. Daino, *Unifying carrier and cloud networks: Problem statement and challenges*, IETF Draft, 2014 (cit. on p. 11).

[156] T. Taleb, "Toward carrier cloud: Potential, challenges, and solutions," *Wireless Communications*, vol. 21, no. 3, pp. 80–91, 2014 (cit. on p. 11).

[157] T. Taleb, A. Ksentini, and A. Kobbane, "Lightweight mobile core networks for machine type communications," *IEEE Access*, vol. 2, pp. 1128–1137, Sep. 2014 (cit. on p. 12).

[158] R. Toghraee, *Comparing the Broadcom silicons used in datacenter switches*, Jun. 2016. [Online]. Available: `https://www.linkedin.com/pulse/comparing-broadcom-silicons-used-datacenter-switches-reza-toghraee` (cit. on p. 106).

[159] F. Travostino, P. Daspit, L. Gommans, C. Jog, C. De Laat, J. Mambretti, I. Monga, B. Van Oudenaarde, S. Raghunath, and P. Y. Wang, "Seamless live migration of virtual machines over the MAN/WAN," *Future Generation Computer Systems*, vol. 22, no. 8, pp. 901–907, 2006 (cit. on p. 28).

[160] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel virtualization technology," *Computer*, vol. 38, no. 5, pp. 48–56, 2005 (cit. on p. 7).

[161] K. Vaidyanathan, W. Huang, L. Chai, and D. K. Panda, "Designing efficient asynchronous memory operations using hardware copy engine: A case study with I/OAT," in *International Parallel & Distributed Processing Symposium (IPDPS)*, IEEE, 2007, pp. 1–8 (cit. on p. 20).

[162] S.-Y. Wang and I.-Y. Lee, "MiniReal: A real SDN network testbed built over an SDN bare metal commodity switch," in *International Conference on Communications (ICC)*, IEEE, 2017, pp. 1–6 (cit. on pp. 33, 95).

[163] S.-Y. Wang, S.-Y. Liu, and C.-L. Chou, "Design, implementation and performance evaluation of software openflow flow counters in a bare metal commodity switch," in *International Symposium on Computers and Communication (ISCC)*, IEEE, 2016, pp. 651–656 (cit. on pp. 33, 95).

[164] J. Whiteaker, F. Schneider, and R. Teixeira, "Explaining packet delays under virtualization," *SIGCOMM Computer Communication Review (CCR)*, vol. 41, no. 1, pp. 38–44, Jan. 2011 (cit. on p. 19).

[165] D. Williams and H. Jamjoom, "Cementing high availability in OpenFlow with RuleBricks," in *SIGCOMM Workshop on Hot topics in software defined networking (HotSDN)*, ACM, 2013, pp. 139–144 (cit. on p. 27).

[166] L. Xia, Q. Wu, D. King, H. Yokota, and N. Khan, *Requirements and use cases for virtual network functions*, IETF Draft, ID: draft-xia-vnfpool-use-cases-02, Nov. 2014 (cit. on p. 11).

[167] H. Xie, Y. Li, J. Wang, D. Lopez, T. Tsou, and Y. Wen, "vRGW: Towards network function virtualization enabled by software defined networking," in *International Conference on Network Protocols (ICNP)*, IEEE, 2013 (cit. on p. 12).

[168] K. Yamazaki, T. Osaka, S. Yasuda, and A. Miyazaki, "Accelerating SDN/NFV with transparent offloading architecture," in *Open Networking Summit (ONS)*, USENIX, 2014 (cit. on p. 31).

[169] D. Yin, D. Unnikrishnan, Y. Liao, L. Gao, and R. Tessier, "Customizing Virtual Networks with Partial FPGA Reconfiguration," *SIGCOMM Computer Communication Review (CCR)*, vol. 41, no. 1, pp. 125–132, 2011 (cit. on pp. 77, 88, 131).

[170] Y. Zaki, L. Zhao, C. Goerg, and A. Timm-Giel, "LTE wireless virtualization and spectrum management," in *Wireless and Mobile Networking Conference (WMNC)*, IFIP, 2010 (cit. on p. 12).

[171] B. Zhang, X. Wang, R. Lai, L. Yang, Z. Wang, Y. Luo, and X. Li, "Evaluating and optimizing I/O virtualization in kernel-based virtual machine (KVM)," in *International Conference on Network and Parallel Computing (NPC)*, 2010, pp. 220–231 (cit. on p. 22).

[172] F. Zhang, G. Liu, X. Fu, and R. Yahyapour, "A survey on virtual machine migration: Challenges, techniques, and open issues," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 2, pp. 1206–1243, 2018 (cit. on pp. 3, 28).

[173] X. Zhang, X. Shao, G. Provelengios, N. K. Dumpala, L. Gao, and R. Tessier, "Scalable network function virtualization for heterogeneous middleboxes," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2017, pp. 219–226 (cit. on p. 32).

[174] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "NetFPGA SUME: Toward 100 gbps as research commodity," *Micro*, vol. 34, no. 5, pp. 32–41, 2014 (cit. on pp. 23, 24).

Note: All web references have been checked in June 2018.

A APPENDIX

A.1 PROOF OF EQUAL MIGRATION DURATION USING DIFFERENT PRIORITIZATION SCHEMES

This section provides an extended derivation of the prioritization problem equations described in Section 4.5.6.2.

A.1.1 *Scenario 2 and Scenario 1*

In the following, we derive that $d_2 = d_1 = \frac{S}{c-r}$. In the second scenario ② in Figure 15, the time needed to empty the buffer and transfer the remaining resync data can be modeled as an infinite number of *rounds*. The sum of all of these rounds results in the total migration time.

b_0 is the time needed to transfer the snapshot in the first round using the entire link capacity. b_1 is the time required to transfer the buffered resync stream, the buffer contents have the size $b_0 \cdot r$ after the first round has been finished. These contents now can be transferred in the following using the entire link capacity c. However, during the latter transfer, the resync stream continues, and must be further buffered and transferred thereafter. This results in an infinte sequence of rounds with individual durations, given in Equation 17.

$$b_0 = \frac{S}{c}, \qquad b_1 = b_0 \cdot \frac{r}{c}, \qquad b_{n+1} = b_n \cdot \frac{r}{c}. \tag{17}$$

The sum of the individual durations required for the infinite number of rounds results in the total migration duration d_2 (Equation 18).

$$d_2 = \sum_{i=0}^{\infty} \frac{S}{c} \cdot \left(\frac{r}{c}\right)^i, \quad a_0 = \frac{S}{c}, \quad q = \frac{r}{c}. \tag{18}$$

Based on our assumption that the resync data rate must not exceed the link capacity ($r < c$), the geometric series converges to the migration duration value (d_1) of Scenario ① (Equation 19). Thus, the migration duration in Scenario ② is equal to Scenario ①.

$$\begin{aligned} d_2 &= \frac{\frac{S}{c}}{1-\frac{r}{c}} = \frac{S}{c \cdot (1-\frac{r}{c})} \\ &= \frac{S}{c - c \cdot \frac{r}{c}} \qquad = \frac{S}{c-r} = d_1. \end{aligned} \tag{19}$$

A.1.2 *Scenario 3 and Scenario 1*

In the following, we derive that $d_2 = d_1 = \frac{S}{c-r}$. The third scenario ③ in Figure 15 is similar to the second one. The major difference is that the resync stream is partially transferred with rate r_a, and partially buffered with the rate $r - r_a$. Thus, the snapshot is transferred with the rate $C - r_a$ only , and b_0 takes longer. As the resync stream is only partially buffered, the time of the next round b_1 is smaller instead. While using the entire capacity c to transmit the buffered resync data during the second round, the resync stream arriving during this round must be completely buffered again and sent later, which repeats infintely. The infinte sequence of rounds is given in Equation 20.

$$b_0 = \frac{S}{c - r_a}, \qquad b_1 = b_0 \cdot \frac{r - r_a}{c}, \qquad n > 0 \Rightarrow b_{n+1} = b_n \cdot \frac{r}{c}. \tag{20}$$

The sum of the individual durations required for the infinite number of rounds results in the total migration duration d_3 (Equation 21).

$$\begin{aligned} d_3 &= \frac{S}{c - r_a} + \sum_{i=0}^{\infty} \frac{S}{c - r_a} \cdot \frac{r - r_a}{c} \cdot \left(\frac{r}{c}\right)^i \\ &= \frac{S}{c - r_a} + \sum_{i=0}^{\infty} \frac{S \cdot (r - r_a)}{c \cdot (c - r_a)} \cdot \left(\frac{r}{c}\right)^i, \quad a_0 = \frac{S \cdot (r - r_a)}{c \cdot (c - r_a)}, \quad q = \frac{r}{c}. \end{aligned} \tag{21}$$

Based on our assumption that the resync data rate must not exceed the link capacity ($r < c$), the geometric series converges to the migration duration value (d_1)

of Scenario ① and the migration duration value (d_2) of Scenario ② (Equation 22). Thus, the migration duration in Scenario ③ is equal to Scenario ①.

$$
\begin{aligned}
d_3 &= \frac{S}{c-r_a} + \frac{\left(\frac{S\cdot(r-r_a)}{c\cdot(c-r_a)}\right)}{1-\frac{r}{c}} \\
&= \frac{S}{c-r_a} + \frac{S\cdot(r-r_a)}{c\cdot(c-r_a) - \frac{r}{c}\cdot c\cdot(c-r_a)} \\
&= \frac{S}{c-r_a} + \frac{S\cdot(r-r_a)}{c\cdot(c-r_a) - r\cdot(c-r_a)} \\
&= \frac{S}{c-r_a} + \frac{S\cdot(r-r_a)}{(c-r)\cdot(c-r_a)} \\
&= \frac{S\cdot(c-r)}{(c-r_a)\cdot(c-r)} + \frac{S\cdot(r-r_a)}{(c-r_a)\cdot(c-r)} \\
&= \frac{S\cdot(c-r) + S\cdot(r-r_a)}{(c-r_a)\cdot(c-r)} \\
&= \frac{S\cdot(c-r+r-r_a)}{(c-r_a)\cdot(c-r)} \\
&= \frac{S\cdot(c-r_a)}{(c-r_a)\cdot(c-r)} \\
&= \frac{S}{c-r} = d_2 = d_1.
\end{aligned}
\tag{22}
$$

A.2 DUAL IMPLEMENTATION OF THE DPI VNF ON FPGA AND COMMODITY CPU

> The ideas and findings presented in this section have been previously published by the author of this thesis [118], and are based on findings of a master's thesis (B. Rudolph [144]) supervised by the author. The following section contains citations from the author's publication [118] which are not explicitly marked. For more information, refer to Appendix B. Cited figures are explicitly marked.

Our system model operates on a state-of-the-art NFV infrastructure optimized for high network performance – with certain differences or constraints: *Requirements*

- Performance-critial parts of network function implementations are not only available for commodity CPUs, but are also available in an FPGA gate logic format.
- The network function and the NFVI controller can determine the current performance demand of a network function, either with the help of the vNF implementation or autonomously, and predict this demand with a given confidence.
- A pool of FPGA resources with high-bandwidth transceivers accessible over the network can be flexibly provisioned, programmed, and released by network functions.

- The flexible underlying software-defined network is able to redirect traffic to either an FPGA or a commodity CPU resource, based on the aforementioned controller's decision.

NF behavior

To conduct the evaluation, we have implemented an NF solving a typical deep packet inspection (DPI) task. This task is on the one hand too complex to solve it by OpenFlow, on the other hand it is simple enough to implement it on an FPGA (see Section 5.1). The semantics of the DPI NF are as follows: First, the NF detects received Transmission Control Protocol (TCP) packets and determines the start of their payload, hereby it also takes care of TCP options (which is non-trivial compared to fixed-header matching like in Internet Protocol (IP)). The NF then checks if the payload contains a configured byte pattern, and increments a counter if the specific byte pattern has been matched. In either case, the packet is sent to an egress port for performance testing.

A.2.0.1 *FPGA Implementation*

Implementation platform

For the FPGA implementation, we have used Vivado High-Level Synthesis (HLS) and mainly followed the XAPP1209 reference design [84] to achieve greater interoperability and reusability of our implementation. In the reference design, network functions can be developed independently as building blocks, using an *AXI4-Stream* bus interface to connect to a dataplane. The interface has a width of 64 bit and operates in packet mode by using auxiliary signals delimiting the start and end of a packet. With the default clock set to $156,25 MHz$, an NF implementation can thus achieve a theoretical maximum throughput of 10Gbit/s over only one interface instance. The AXI interface is part of the *ARM Advanced Microcontroller Bus Architecture (AMBA)* specification[1], is relatively simple, and is a candidate for a common dataplane interface for hardware-implemented vNF instances.

Implementation details

Xilinx's 10GE Medium Access Control (MAC) Subsystem is used to receive packets from the GTH transceivers and pass them to the following components via AXI4-Stream. The first component is a *Parser* module, included in the reference design. It includes an Address Resolution Protocol (ARP) responder and Internet Control Message Protocol (ICMP) server which may be used in future designs, but are not relevant for our transparently-processing NF at the moment. To support multiple network functions on a single physical port, we have added a filter component to the Parser module, distributing packets to different network functions (including the future state migration module for vNFs). Up to now, the filter module operates based on the ethertype, in the future it may be adapted to match any other fixed-position header fields (e.g. the destination MAC address).

Support for multiple NFs on an FPGA

A *Merge* module combines the packet streams of multiple network functions for output on a physical interface by the MAC subsystem. Its main component is a round-robin arbiter waiting for new packets on the vNF output interfaces. As usual in AXI4-Stream pipelines, our NF is implemented as a finite state machine operating on 64-bit segments of the packet in every clock step. If the Data offset

1 https://www.arm.com/products/system-ip/amba-specifications

field for the TCP segment's payload is found to be beyond the 64-bits limit, the operation is carried to the next clock cycle and so on.

Programmability and Limitations

The FPGA system design with its modular components is synthesized and implemented as a stand-alone bitfile before transfer to the NetFPGA module. Therefore, network functions on the same chip cannot be re-arranged on-the-fly at the moment, however features like Partial Reconfiguration [169] could be integrated in future versions.

A.2.0.2 *Commodity CPU Implementation*

Implemented using a LuaJIT wrapper

To achieve state-of-the-art commodity CPU performance for the x86-based vNF implementation, we have chosen DPDK with the LuaJIT(Just-In-Time) wrapper from the Moongen project[2] as the development platform. LuaJIT is a just-in-time compiler for the Lua programming language, which has been used by Moongen to be able to customize workload generator scripts using a convenient scripting language, while exploiting the performance potential of DPDK. We have re-used and extended the DPDK wrapper of the project to implement our DPI NF in Lua. The implemented functional behavior is targeted to be equal to the FPGA-based implementation.

A.3 ELASTICITY EVALUATION

Listing A.1: Provisioning algorithm to achieve a cost-optimal fitting. Returns a tuple (fpga, cpu), which can process the load with optimal costs. The "liquidcpu" option corresponds to the PerMbit scenario option. With mode=1 or mode=2, only FPGAs or CPUs are provisioned.

```
def get_fitting_2(load, mode=0, liquidcpu=False):

  if mode == 1:
    #FPGA-only
    return (int(math.ceil(load/max_load_fpga)), 0)

  maxfpgas = 0 if mode==2 else int(math.ceil(load/max_load_fpga))

  bestfit = False
  bestfitcosts = -1

  for i in range (0, maxfpgas+1):
    loadleft = load - i*max_load_fpga
    if not liquidcpu:
      cpus = int(math.ceil(loadleft/max_load_cpu)) if loadleft > 0 else 0
    else:
      cpus = loadleft/max_load_cpu if loadleft > 0 else 0

    costs = float(i) + rel_cost_cpu_fpga*cpus
    if not bestfit or costs < bestfitcosts:
      bestfit = (i, cpus)
      bestfitcosts = costs
```

2 Emmerich et al. [37], https://github.com/emmericp/MoonGen

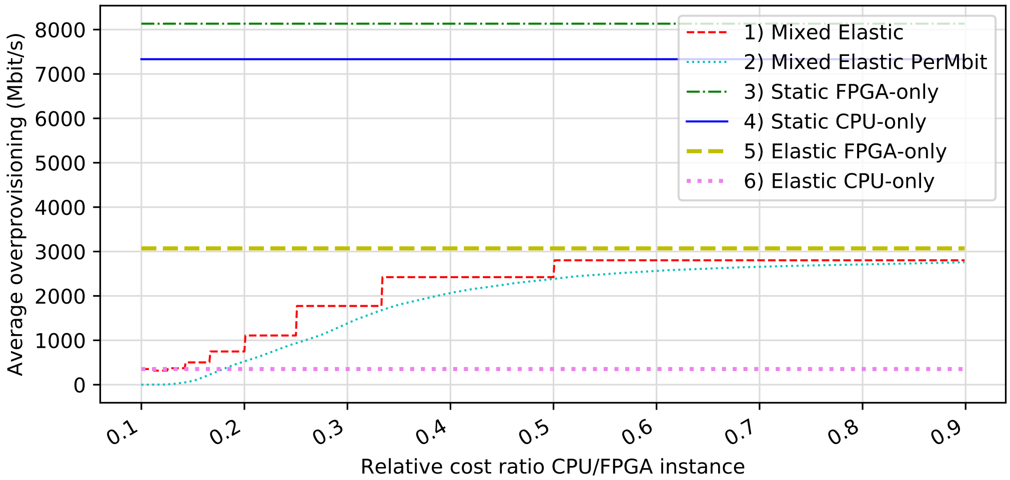


Figure 50: The average overprovisioning, given in megabits per second. Depicted for different relative cost ratios (CPU/FPGA), other parameters according to our reference parameter set (Table 1).

```
return bestfit
```

A.4 BARE-METAL BRAS IMPLEMENTATION - FLOW MODELS AND CONFIGURATION

Listing A.2: Example initial JSON configuration file, defining three services: Internet (VLAN 4) and IPTV (VLAN 8). The internet service uses the core port 7 (untagged), and IPTV uses the core port 8 (tagged). Subscribers and their ports/VLANs are configured at runtime. [119]

```
{
  "configuration": {
    "global_info": {
      "_service_list": [
        {
          "_srv_name":"Internet",
          "_srv_subnet":"100.50.1.0/24",
          "_srv_core_port":"7",
          "_srv_vlan_tag":"4",
          "_srv_next_hop_mac":"00:1b:21:1c:e7:4b",
          "_srv_untagged_flag":"1",
          "_srv_gateway_ipv4_addr":"100.50.1.254",
          "_srv_gateway_ipv4_mask":"255.255.255.0"
        }, {
          "_srv_name":"IPTV",
          "_srv_subnet":"192.168.2.0/24",
          "_srv_core_port":"8",
          "_srv_vlan_tag":"8",
          "_srv_next_hop_mac":"00:30:48:8a:cd:d0",
          "_srv_untagged_flag":"0",
          "_srv_gateway_ipv4_addr":"192.168.2.254",
          "_srv_gateway_ipv4_mask":"255.255.255.0"
        } ],
```

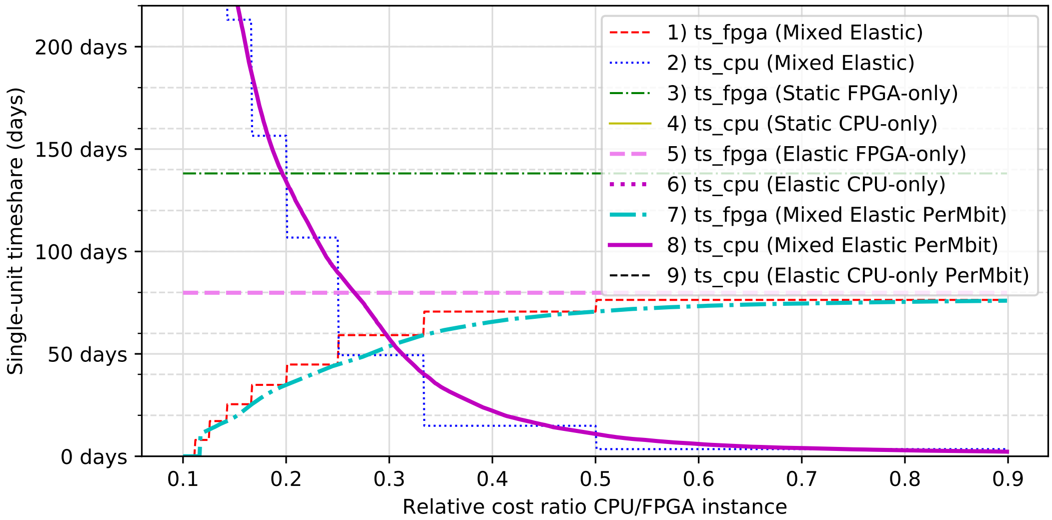


Figure 51: The timeshares of the instances (ts_fpga, ts_cpu) in days. Depicted for different relative cost ratios (CPU/FPGA), other parameters according to our reference parameter set (Table 1).

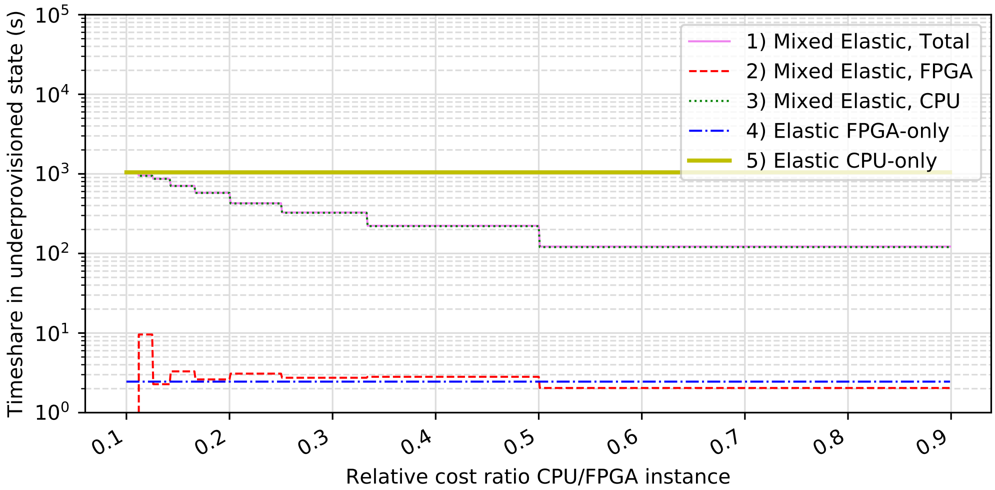


Figure 52: The timeshare of an instance type being in an underprovisioned state, given in seconds. Depicted for different relative cost ratios (CPU/FPGA), other parameters according to our reference parameter set (Table 1).

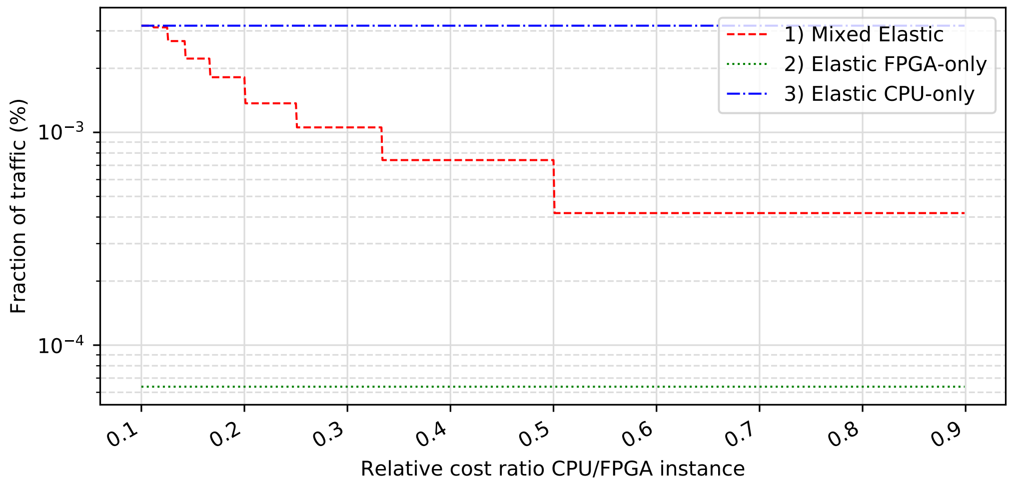


Figure 53: The fraction of traffic which cannot be processed because of underprovisioning, given in percent. Depicted for different relative cost ratios (CPU/FPGA), other parameters according to our reference parameter set (Table 1).

```
    "_ipv6_prefix_length":"64",
    "_bras_sub_mac":"52:54:00:00:00:01",
    "_bras_core_mac":"52:54:00:00:00:02"
   }
  }
}
```

Listing A.3: Example runtime configuration of a subscriber. [119]

```
{"_cmd":"adduser","_arg":{
        "_subsc_id":"1002",
        "_username":"john_doe",
        "_password":"12345",
        "_port_num":"5",
        "_vlan_id":"11",
        "_service_tags":[ "4", "8"]
}}
```

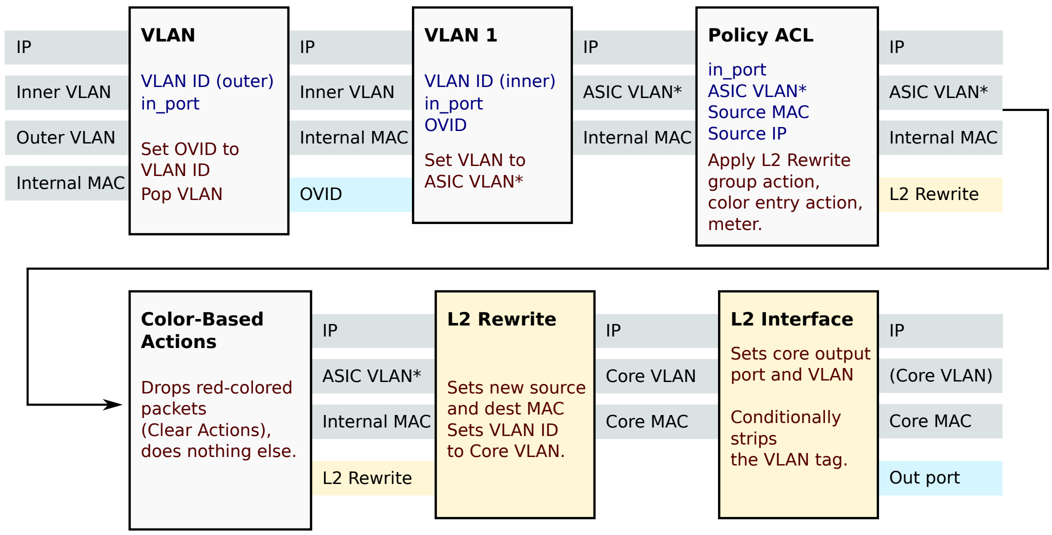


Figure 54: OF-DPA flow model of the double-tagged mode, upstream direction. [119]

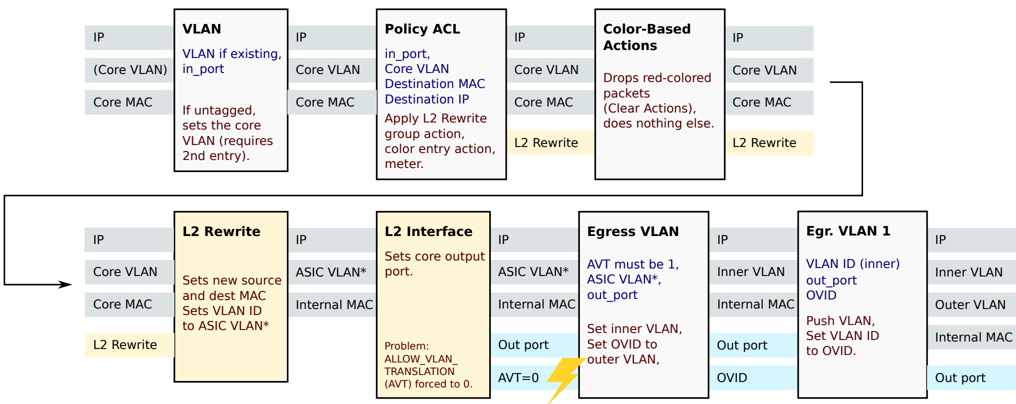


Figure 55: OF-DPA flow model of the double-tagged mode, downstream direction. [119]

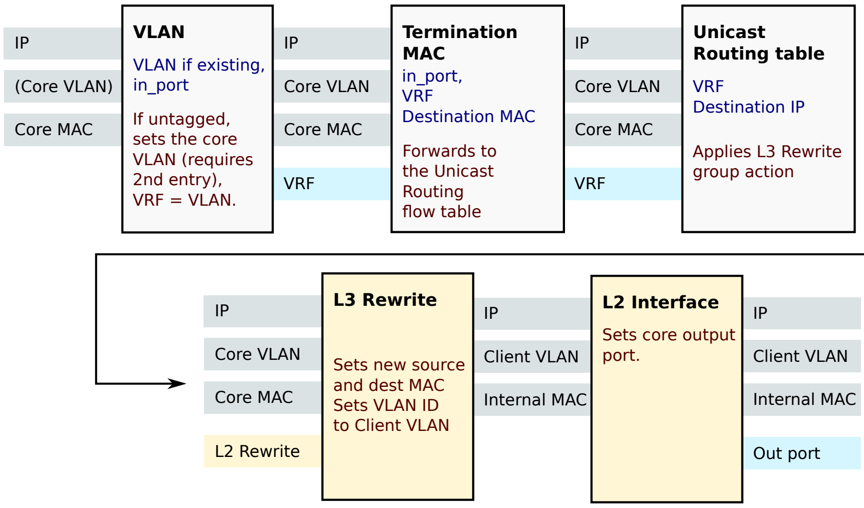


Figure 56: OF-DPA flow model of the single-tagged mode when using the Unicast Routing table, downstream direction. [119]

LIST OF FIGURES

LIST OF TABLES

LISTINGS

GLOSSARY

5G	fifth-generation wireless systems. 12, 33
ACL	access control list. 98, 103, 105
AES	Advanced Encryption Standard. 41
API	application programming interface. 4, 14, 22, 25, 43, 57, 60, 95, 97, 99, 102
ARP	Address Resolution Protocol. 11, 41, 61, 105, 106, 130
ASIC	application-specific integrated circuit. 3, 13, 14, 23–25, 60, 77, 95–97, 99, 101, 103, 107, 109
BNG	broadband network gateway. 12
BRAS	broadband remote access server. 5, 6, 12, 95, 98–106, 108, 109, 139, 140
CDN	content delivery network. 13
CORD	Central Office Re-Architectured as a Datacenter. 11, 33
CPE	customer premises equipment. 101, 102, 104
CPU	central processing unit. 3–5, 7, 8, 13, 16, 18, 20–25, 31, 32, 38, 52, 53, 56–58, 65, 75–95, 97, 99, 107–109, 129–134, 138–140
DC	datacenter. 1–3, 5, 35, 36, 42, 76, 107, 108
DDoS	distributed denial of service. 13, 77
DHCP	Dynamic Host Configuration Protocol. 9, 11
DMA	Direct Memory Access. 20, 21, 54
DPDK	Data Plane Development Kit. 21, 22, 35, 56–59, 66, 68, 69, 78, 82, 107, 131
DPI	deep packet inspection. 13, 81, 82, 94, 130, 131
DSL	Digital Subscriber Line. 76, 98, 109, 142

DSLAM	DSL access multiplexer. 98, 99, 102, 104
EAPOL	Extensible Authentication Protocol over local area network (LAN). 101
eNodeB	Evolved NodeB. 42
EPC	Evolved Packet Core. 42
ETSI	European Telecommunications Standards Institute. 8, 10, 14, 15, 32, 137
FPGA	field-programmable gate array. 3, 7, 23, 24, 31, 32, 75–95, 108, 129–134, 138–140
GPRS	General Packet Radio Service. 42
HDL	hardware description language. 23, 31, 77, 79
HWA	hardware acceleration. 1, 3–6, 31, 32, 76–81, 107–109, 138
I/O	input/output. 10, 11, 19–23, 28, 50, 54, 56, 66, 68, 78, 97, 137
ICMP	Internet Control Message Protocol. 9, 61, 130
IDS	intrusion detection system. 17, 28, 41
IEEE	Institute of Electrical and Electronics Engineers. 101, 105
IETF	Internet Engineering Task Force. 11
IP	Internet Protocol. 1, 9, 12–14, 17, 18, 22, 40–42, 58, 61, 67, 68, 77, 81, 98, 99, 102, 103, 106, 130, 142, 144
IPTV	IP Television. 18, 99
ITU	International Telecommunication Union. 18
KVM	Kernel-based Virtual Machine. 56, 65, 82, 107
L2	Data Link Layer. 19, 21, 66, 97, 98, 103
L3	Network Layer. 76, 97, 98
LAN	local area network. 19, 61, 97, 101, 142, 144
LTE	Long-Term Evolution. 12, 42, 109
MAC	Medium Access Control. 21, 41, 98, 99, 101, 102, 130

MPLS multi-protocol label switching. 98, 103
MSS maximum segment size. 58
MTU maximum transmission unit. 68

NAT network address translation. 11, 12, 14, 28, 29, 40–42, 61, 68, 70–73, 77, 137, 138
NF network function. 1–5, 8–14, 16–20, 22, 24, 28–31, 33, 37–46, 49, 54, 56, 61–63, 65, 68–78, 82, 85, 86, 88, 93–95, 107–109, 130, 131, 137, 138
NFV network functions virtualization. 1–8, 10–19, 23, 27–29, 31–33, 35, 44–46, 48, 50, 51, 56, 64–66, 76–78, 80, 88, 95, 107–109, 129, 137, 149
NIC network interface controller. 20–22, 65, 66
NPU network processing unit. 77–80

OF-DPA OpenFlow Data Plane Abstraction. 11, 14, 25, 33, 56, 60, 66, 97–99, 102–107, 135, 136, 139, 140
OLT optical line termination. 99, 100
OS operating system. 7, 8, 19, 22, 96, 97
OVS Open vSwitch. 19, 143
OVSDB Open vSwitch (OVS) Database. 19, 25

P4 Programming Protocol-Independent Packet Processors. 7, 24, 76, 81, 109
PCI Peripheral Component Interconnect. 21, 22, 52, 66, 75, 79, 143
PCI-PT Peripheral Component Interconnect (PCI) passthrough. 21, 66, 82
PPP Point-to-Point Protocol. 12, 76, 108, 143
PPPoE PPP over Ethernet. 12, 76, 79, 99, 108

QoS quality of service. 19, 97

RADIUS Remote Authentication Dial-In User Service. 101
RAM random access memory. 30
RFC Request For Comments. 8, 9, 82
RTP Real-time Transport Protocol. 77
RTT round-trip time. 67–70, 72, 138

S-GW Serving Gateway. 42, 61
SBC session border controller. 12, 77
SCSI Small Computer Systems Interface. 8
SDN software-defined networking. 2, 7, 10–14, 16, 17, 19, 25, 28, 29, 31, 44–50, 56, 60, 66, 76–78, 81, 82
SIP Session Initiation Protocol. 12
SLA service level agreement. 70, 84
SliM Seamless Instance Migration. 5, 6, 35, 44–46, 48–50, 56–58, 60, 61, 64–74, 88, 107–109, 137, 138, 149
SNMP Simple Network Management Protocol. 11
SR-IOV Single-Root I/O Virtualization. 21, 78
SSH Secure Shell. 41, 96

TCP Transmission Control Protocol. 13, 22, 36, 40, 41, 47, 52, 56–58, 61, 62, 64, 68, 130, 131
TLV type-length-value. 57, 58

UDP User Datagram Protocol. 13, 40–42, 61, 67, 68

VF SR-IOV virtual function. 21, 66
VIE vNF instance environment. 44–46, 56, 57
VIM virtualized infrastructure manager. 14
VLAN Virtual LAN. 19, 21, 44, 97–99, 101–106
VM virtual machine. 1, 2, 4, 8, 14, 18–22, 27–31, 35, 37, 40, 43, 44, 46, 50, 54–56, 60, 65–67, 81, 82, 87, 104, 109, 137, 144
VMDq VM Device Queues. 21, 22
vNF virtualized network function. 1–3, 10, 14, 18, 27, 28, 31, 35–37, 39, 43–54, 56, 57, 61, 66, 67, 75, 78–81, 86, 87, 108, 109, 129–131, 138, 144
VoIP Voice-over-IP. 18, 68, 77, 99
VPN virtual private network. 41, 77
VXLAN Virtual Extensible LAN. 97, 98

WAN wide area network. 3–5, 28–30, 35–37, 50, 52, 53, 55, 67–69, 101, 107, 108

B SELF-CITATIONS

A minor part of the thesis contains citations from the author's own work which are not explicitly marked. The following list contains sections which have been cited and have been adapted to the structure and terminology of this thesis, along with their original source.

- Section 2.11.1 contains citations from [119],
- Section 5.1.1 contains citations from [122],
- Section 5.1.2 contains citations from [122],
- Section 5.1.3 contains citations from [122],
- Section 5.2 and its subsections contain citations from [122],
- Section 6.1 and its subsections contain citations from [119],
- Section 6.2 and its subsections contain citations from [119],
- Section 6.3 and its subsections contain citations from [119],
- Section A.2 and its subsections contain citations from [118],

Figures, listings and tables which have been cited are explicitly marked.

C

PUBLICATIONS OF THE AUTHOR

C.1 CONFERENCE PAPERS AND JOURNAL ARTICLES (MAIN AUTHOR)

[118] L. Nobach, B. Rudolph, and D. Hausheer, "Benefits of conditional FPGA provisioning for virtualized network functions," in *International Conference and Workshops on Networked Systems (NetSys)*, IEEE, Mar. 2017, pp. 1–6 (cit. on pp. 75, 82, 86, 87, 129, 145).

[119] L. Nobach, J. Blendin, H.-J. Kolbe, G. Schyguda, and D. Hausheer, "Bare-metal switches and their customization and usability in a carrier-grade environment," English, in *Conference on Local Computer Networks (LCN)*, IEEE, Oct. 2017, pp. 649–657 (cit. on pp. 7, 95, 96, 100–106, 132, 134–136, 140, 141, 145).

[121] L. Nobach and D. Hausheer, "Towards decentralized, energy- and privacy-aware device-to-device content delivery," in *Monitoring and Securing Virtualized Networks and Services*, Springer, 2014, pp. 128–132.

[122] L. Nobach and D. Hausheer, "Open, elastic provisioning of hardware acceleration in NFV environments," in *International Conference and Workshops on Networked Systems (NetSys)*, IEEE, Mar. 2015, pp. 1–5 (cit. on pp. 15, 31, 75, 79, 80, 145).

[123] L. Nobach and D. Hausheer, "PrivateShare: Measuring device-to-device user behavior and transmission quality," in *Network Operations and Management Symposium (NOMS)*, IEEE/IFIP, Apr. 2016, pp. 830–833.

[124] L. Nobach, O. Hohlfeld, and D. Hausheer, "New kid on the block: Network functions virtualization: From big boxes to carrier clouds (editorial)," *SIGCOMM Computer Communication Review (CCR)*, Jul. 2016 (cit. on pp. 7, 20).

[125] L. Nobach, Y. L. Louédec, and D. Hausheer, "Evaluating device-to-device content delivery potential on a mobile ISP's dataset," in *International Conference on Network and Service Management (CNSM)*, Nov. 2015, pp. 301–309.

[126] L. Nobach, I. Rimac, V. Hilt, and D. Hausheer, "SliM: Enabling efficient, seamless NFV state migration," in *International Conference on Network Protocols (ICNP)*, IEEE, Nov. 2016, pp. 1–2 (cit. on p. 35).

[127] L. Nobach, I. Rimac, V. Hilt, and D. Hausheer, "Statelet-based efficient and seamless NFV state transfer," English, *Transactions on Network and Service Management (TNSM)*, vol. 14, no. 4, pp. 964–977, Dec. 2017 (cit. on pp. 35, 39, 45, 46, 48, 51, 54, 55, 57, 65, 66, 70–74).

C.2 PATENTS

[142] I. Rimac, L. Nobach, and V. Hilt, *Method for migration of virtual network function*, EP Patent App. EP20,160,290,053, Sep. 2017.

C.3 DEMOS

[120] L. Nobach, J. Blendin, H.-J. Kolbe, G. Schyguda, and D. Hausheer, "RTP packet loss healing on a bare-metal switch (demo)," in *Network Operations and Management Symposium (NOMS)*, Apr. 2018.

C.4 CO-AUTHORED PUBLICATIONS

[94] P. Lieser, N. Richerzhagen, T. Feuerbach, L. Nobach, D. Böhnstedt, and R. Steinmetz, "Take it or leave it: Decentralized resource allocation in mobile networks," in *Conference on Local Computer Networks (LCN)*, IEEE, 2017.

[152] D. Stingl, C. Gross, L. Nobach, R. Steinmetz, and D. Hausheer, "BlockTree: Location-aware decentralized monitoring in mobile ad hoc networks," in *Conference on Local Computer Networks (LCN)*, IEEE, 2013, pp. 373–381.

[153] D. Stingl, C. Gross, J. Rückert, L. Nobach, A. Kovacevic, and R. Steinmetz, "PeerfactSim.KOM: A simulation framework for peer-to-peer systems," in *Conference on High Performance Computing and Simulation (HPCS)*, IEEE, 2011, pp. 577–584.

C.5 OPEN-SOURCE SOFTWARE RELATED TO THIS THESIS

- SliM State Migration Framework: `https://github.com/nokia/SliM`,
- Sloth DPDK-based Latency Generator: `https://github.com/lnobach/sloth-latency-gen`,

D

ACKNOWLEDGMENTS

First of all, I want to thank **Prof. Dr.-Ing. Ralf Steinmetz** for giving me the chance to finish my Ph.D. research in an exceptionally good working atmosphere at KOM, which I can recommend to anyone seeking a place for pursuing a doctor's degree. I thank him that I could profit from his experience obtained from many years in science, and for giving me confidence in my work. In this regard, I also want to thank my direct supervisor **Dr. Boris Koldehofe**. He invested much of his time for giving very valuable and detailed feedback to my dissertation, and kept my back free during the last stretch of my dissertation write-up.

Likewise, I want to thank my supervisor for more than 3 years, **Prof. Dr. David Hausheer**, who not only taught me very valuable scientific writing and publication skills, but also introduced me to the scientific area and community of NFV. I thank him for his strong and continuing support over the past years, especially regarding the many publications we have achieved together.

I also thank **Prof. Dr.-Ing. Wolfgang Kellerer** that he took his time and agreed to be the co-reviewer of my dissertation.

I want to thank my former colleagues at *Nokia Bell Labs*, **Dr.-Ing. Ivica Rimac** and **Dr. Volker Hilt**, they supported me during the conception and implementation phase of SliM, a major contribution of this thesis.

I also want to thank my colleagues at *Deutsche Telekom AG*, **Dr. Hans-Jörg Kolbe**, **Georg Schyguda**, and **Robert Soukup**, for the exciting "D-Nets" collaboration, as well as for the "carrier-grade" advice and feedback to my work.

I want to thank my master's students **Benedikt Rudolph**, **Zhiqiang Qian**, and **Lukas Fey** for their valuable master's thesis work thriving my research topics.

I want to thank all my former colleagues at TU Darmstadt. I especially want to thank **Jeremias Blendin** for the many technical discussions we had about **SDN** and **NFV**, helping me always updating my knowledge with the latest hot stuff in networking. I also want to thank **Ralf Kundel** for continuing some of the research topics I started but could not finish.

Special thanks go to my family, my mother **Christiana**, my father **Eckehard**, and my sister **Josephine**. Although they do not know as much about NFV, they supported me over the entirety of 32 years, and therefore contributed in a very special way to this work.

The last paragraph is dedicated to my fiancée **Janine**, who was supportive in so many ways. As a researcher, you probably know the phases where your evaluation result does not confirm the hypothesis, and after 4 nights of frustrated debugging, you discover that it was caused by typo in line 1321 of `main.c`. Janine helped me through these phases by always standing by me and my work.

D.1 FUNDING

This work has been supported in parts by the German Research Foundation (DFG) within the Collaborative Research Center (CRC) 1053 - MAKI, by the project "Dynamic Networks (D-Nets) 3-6" in collaboration with Deutsche Telekom AG, by a Ph.D. internship at Nokia Bell Labs, and by the European Union (FP7/#317846, SmartenIT and FP7/#318398, eCousin).

E CURRICULUM VITÆ

E.1 PERSONAL INFORMATION

Last Name	Nobach
Given Names	Leonhard Wolfgang
Date of Birth	May 11th, 1986
Place of Birth	Kassel
Nationality	German

E.2 EDUCATIONAL BACKGROUND

Oct. 2009 – May 2012	**Technische Universität Darmstadt** Computer Science (Master of Science) Minor Field of Study: IT Management (Business Administration, Law) Master's Thesis: "Location-Aware Data Aggregation in Mobile P2P Networks"
Oct. 2005 – Jul. 2009	**Technische Universität Darmstadt** Computer Science (Bachelor of Science) Bachelor's Thesis: "Assessing Quality of Peer-to-Peer Search Overlays"
1996 - Jul. 2005	**Wilhelmsgymnasium Kassel** Abitur (General Qualification for University Entrance)

E.3 PROFESSIONAL EXPERIENCE

Since May 2017	**Technische Universität Darmstadt - Fachgebiet Multimedia Kommunikation (KOM)** – Scientist, Doctoral Candidate Research in the area of Network Functions Virtualization (NFV). Project "Dynamic Networks 7" together with Deutsche Telekom AG.
Jun. 2015 – Jan. 2016	**Nokia Bell Labs**, Stuttgart – Ph.D. Intern Design and implementation of the SliM state migration mechanism (Chapter 4).
Oct. 2013 – Apr. 2017	**Technische Universität Darmstadt - Fachgebiet Entwurfsmethodik für Peer-to-Peer-Systeme (PS)** – Scientist Research and teaching in the area of network functions virtualization (NFV), software-defined networking (SDN) and device-to-device (D2D) content delivery.
Oct. 2012 – Sep. 2013	**Dimension Data Germany AG & Co. KG** – Technology Associate Trainee programme as an engineer for network infrastructure and security solutions.
Nov. 2008 – Jun. 2011	**KOM – Multimedia Communications Lab, TU Darmstadt** – Student Assistant Maintenance and development of PeerfactSim.KOM, a simulator for large-scale peer-to-peer network protocols. Provided support for students which have written their theses in this field of study.

E.4 AWARDS

Jul. 2009	**KOM-Fördergesellschaft e.V.** Best Bachelor's Thesis of the Year 2009

E.5 REVIEWER ACTIVITY

- IEEE Conference on Local Computer Networks (LCN) *2015, 2016, 2017, 2018,*
- IFIP Networking *2015, 2016, 2017,*
- International Conference on Autonomous Infrastructure, Management and Security (AIMS) *2016, 2017,*
- IFIP/IEEE International Symposium on Integrated Network Management (IM) *2015, 2017,*

- International Conference on Network and Service Management (CNSM) *2015*, *2016*,
- Network Operations and Management Symposium (NOMS) *2016*,
- ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc) *2016*,
- International Journal of Network Management (IJNM) *2015*.

E.6 SUPERVISED STUDENT THESES

- Benedikt Rudolph. *Evaluating Benefits of Elastic FPGA Accelerator Provisioning in NFV Environments.* Master's Thesis, 2016.
- Zhiqiang Qian. *Traffic Distribution for Heterogeneous Hardware Processing in NFV Environments.* Master's Thesis, 2016.
- Lukas Fey. *A Migration-Cost-Aware Evaluation Framework for VNF Placement Strategies.* Master's Thesis, 2018.

F

DECLARATION

ERKLÄRUNGEN LAUT PROMOTIONSORDNUNG

§ 8 Abs. 1 lit. c PromO

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

§ 8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

§ 9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.

§ 9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, November 2018

Leonhard Nobach

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

Final Version as of November 4, 2018 (`classicthesis` version 1.0).