



Survey of Performance Acceleration Techniques for Network Function Virtualization

Leonardo Linguaglossa, Stanislav Lange, Salvatore Pontarelli, Gábor Rétvári, D. Rossi, Thomas Zinner, Roberto Bifulco, Michael Jarschel, Giuseppe Bianchi

► To cite this version:

Leonardo Linguaglossa, Stanislav Lange, Salvatore Pontarelli, Gábor Rétvári, D. Rossi, et al.. Survey of Performance Acceleration Techniques for Network Function Virtualization. Proceedings of the IEEE, 2019, 10.1109/JPROC.2019.2896848 . hal-03022144

HAL Id: hal-03022144

<https://hal.science/hal-03022144v1>

Submitted on 24 Nov 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Survey of Performance Acceleration Techniques for Network Function Virtualization

Leonardo Linguaglossa[†], Stanislav Lange^{*}, Salvatore Pontarelli[§], Gábor Rétvári[¶], Dario Rossi[†], Thomas Zinner^{||}, Roberto Bifulco^{**}, Michael Jarschel^{††}, Giuseppe Bianchi[‡]

[†]Telecom ParisTech, ^{*}University of Würzburg, [‡]CNIT (National Inter-University Consortium for Telecommunications) / University of Rome “Tor Vergata”, [§]Axbryd / CNIT (National Inter-University Consortium for Telecommunications), [¶]MTA-BME Information Systems Research Group, ^{||}Technical University of Berlin, ^{**}NEC Laboratories Europe, ^{††}Nokia Bell Labs

Abstract—The ongoing network softwarization trend holds the promise to revolutionize network infrastructures by making them more flexible, reconfigurable, portable, and more adaptive than ever. Still, the migration from hard-coded/hard-wired network functions towards their software-programmable counterparts comes along with the need for tailored optimizations and acceleration techniques, so as to avoid, or at least mitigate, the throughput/latency performance degradation with respect to fixed function network elements. The contribution of this article is twofold. First, we provide a comprehensive overview of the host-based Network Function Virtualization (NFV) ecosystem, covering a broad range of techniques, from low level hardware acceleration and bump-in-the-wire offloading approaches, to high-level software acceleration solutions, including the virtualization technique itself. Second, we derive guidelines regarding the design, development, and operation of NFV-based deployments that meet the flexibility and scalability requirements of modern communication networks.

Index Terms—NFV, Performance Acceleration, Fast Packet Processing, Offloading, Virtualization.

I. INTRODUCTION

Current communication and networking use cases such as industrial Internet, Internet of Things (IoT), video streaming, and vehicle-to-everything communication confront operators with numerous challenges. On the one hand, an ever growing number of devices and services calls for scalable systems that can cope with the increasing resource demands. On the other hand, the heterogeneity of demands and communication types as well as their temporal dynamics require a high degree of adaptability in order to operate the network in an efficient manner while meeting performance criteria.

A possible solution for overcoming the limitations of today’s network architectures and addressing these challenges is offered by ecosystems that are built around the paradigms of Software Defined Networking (SDN) and Network Functions Virtualization (NFV). While SDN provides a flexible and programmable infrastructure, NFV replaces dedicated hardware middleboxes with software instances that can be deployed, scaled, and migrated in a dynamic fashion. However, using software that runs on common-off-the-shelf (COTS) hardware to replace special-purpose ASIC-based hardware can have a severe performance impact, e.g., in terms of key performance indicators such as throughput and latency, affecting the overall end-to-end application performance. Hence, recent research

efforts focus on various acceleration techniques for Virtual Network Functions (VNFs) that address these performance issues at different levels. These include software-based approaches that optimize packets’ traversal through the software stack after their reception at the network interface card, as well as approaches that offload parts of the processing pipeline to programmable hardware. Furthermore, acceleration techniques can be specialized in particular aspects of the processing pipeline and therefore improve the performance regarding resources such as compute, memory, or storage.

One particular problem that arises in such ecosystems is the choice of an appropriate acceleration technique given a specific combination of network function and virtualization context. There are two main sources of heterogeneity that complicate this task. First, network functions differ significantly w.r.t. the type and intensity of their resource usage, e.g., compute-heavy transcoding operations as opposed to caches that require a large amount of storage I/O. Second, the virtualization stack, on which the corresponding deployment is built, has a large impact on the availability and applicability of acceleration techniques. Figure 1 shows a compact representation of the scenario considered in the remainder of this paper. We provide in the figure a reference to the Sections devoted to the description of the components of this generic architecture, ranging from pure software components that offer a high degree of programmability to pure hardware devices that have low or no programmability. From a top-down point of view, VNFs are software components built on top of one or more host systems, mainly consisting of commodity PCs. Section II focuses on the ecosystem of VNFs, analyzing the most common network functions and identifying some underlying *building blocks*. The host system is in turn made of several components like CPU, memory, and storage that are tightly coupled: as an example, the main memory might be split into multiple Non-Uniform Memory Access (NUMA) nodes, and the computing unit may consist of several processors, attached to different NUMA nodes. The host system represents a layer on top of which software components such as VNFs can be placed: Section III describes existing pieces of software in scope with the NFV philosophy, as well as the most important software acceleration techniques. Finally, in the lower part of this hierarchy, Network Interface Cards (NICs) are responsible for the actual data exchange among host systems and there-

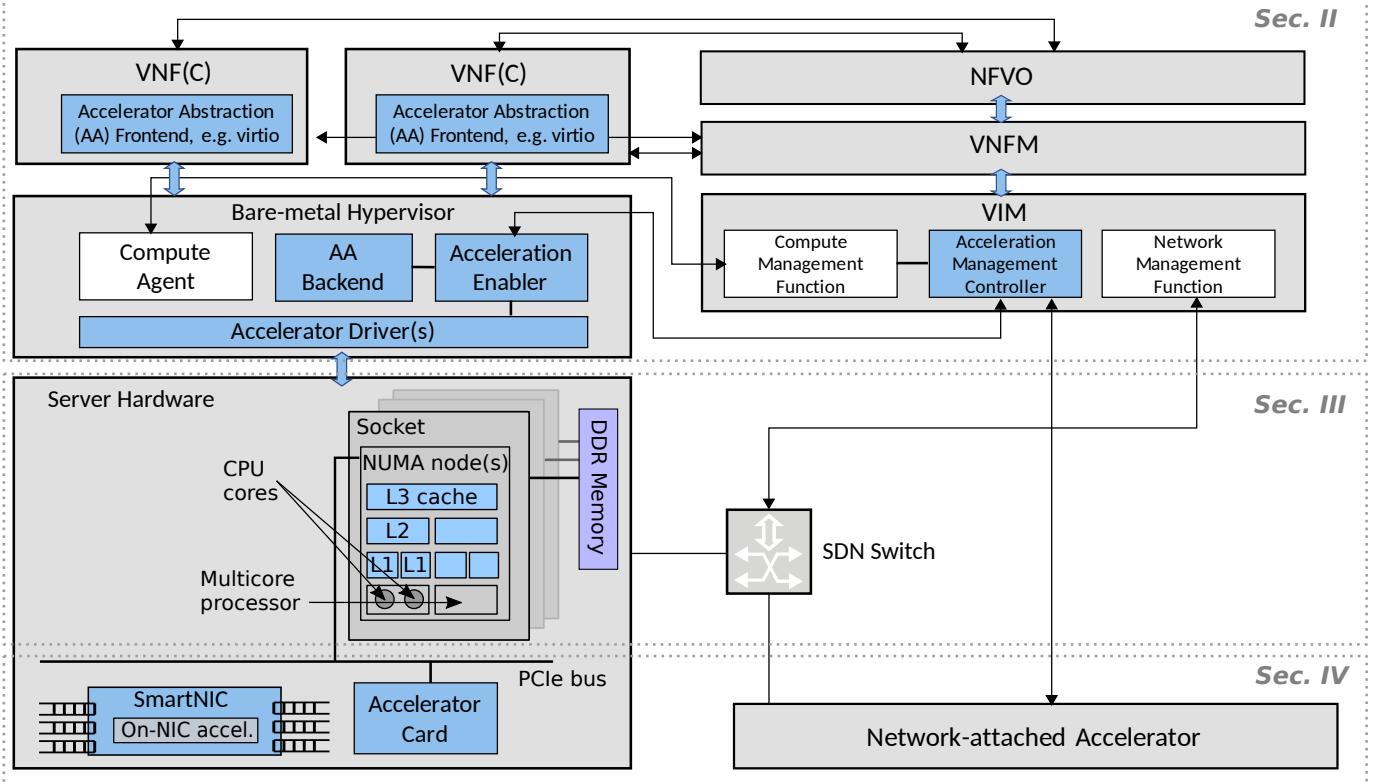


Figure 1: Overview of NFV systems considered in this work. From a top-down perspective, the interaction among the VNFs and the components responsible for the management are explored in Sec.II. The acceleration techniques can be exploited, via the presence of accelerated drivers, after the deployment of VNFs on a server hardware: the most important software acceleration techniques are explored in Sec. III. Finally, hardware acceleration, which can rely on SmartNICs or external accelerators, are described in Sec. IV.

fore, for VNF communication. For some applications, Field-Programmable Gate Arrays (FPGAs) or other programmable hardware can be used to offload part of the computation typically provided by the software components. Although all network function may reside in the top part of Figure 1, the virtualized function implemented may have an effect on a different part of the hardware/software hierarchy. Hence an interesting problem may arise: can all VNFs exploit the same typology of acceleration techniques or there exists some *class* of VNFs for which a specific acceleration technique is better?

We make the following contributions in this article. On the one hand, we provide a comprehensive overview of the NFV ecosystem, i.e., functions that are virtualized, virtualization techniques that provide abstractions for accessing physical resources, and orchestration frameworks that integrate all components into the ecosystem. We further decompose VNFs into functional blocks: the characterization of both existing and novel NFs allows to identify potential performance bottlenecks and, consequently, assess the feasibility and expected benefits coming from acceleration mechanisms. In a similar fashion, we survey acceleration techniques. In the case of software acceleration, we provide an overview of mechanisms, their focus and benefits, as well as various tools and frameworks that utilize them. Hardware acceleration mechanisms are subdivided into standard NICs with basic offloading capabilities,

SmartNICs that offer FPGA-like programmability, as well as domain-specific programming abstractions for NFs. Based on the outlined survey, we derive guidelines for the design, development, and operation of NFV ecosystems that can cope with heterogeneous and dynamic workloads in an efficient manner. In addition to the general view of the NFV ecosystem, we choose one exemplary VNF — the Next Generation Firewall (NGFW) — which leverages a large number of functional blocks, and that thus represents a good example to illustrate how these guidelines translate to a particular use case.

In the remainder of this manuscript we cover the NFV ecosystem (Section II) and survey software (Sections III) and hardware (IV) acceleration techniques. We next present guidelines that are extracted from this survey (Section V) and conclude the work (Section VI).

II. NETWORK FUNCTION VIRTUALIZATION ECOSYSTEM

Numerous areas of networking are targeted by the NFV paradigm [1], [2]. These range from traditional middle-boxes [3] like NATs and firewalls to mobile network functions [4] such as Serving or Packet Gateways (SGW, PGW) and Mobile Management Entities (MME), as well as further extending to customer premises equipment and media transcoders. In this section, we provide a structured overview of the NFV ecosystem in order to cope with the resulting

heterogeneity in terms of requirements for an efficient operation of individual functions as well as management of their coexistence on a shared physical substrate.

First, we break down VNFs into common functional building blocks (Section II-A) to identify consumed resources, possible performance bottlenecks, and therefore opportunities for acceleration. This modular approach allows for a straightforward classification of new functions and their building blocks. Furthermore, guidelines and insights for functions that are composed of the same building blocks can be transferred between functions. Second, we discuss different virtualization options (Section II-B) and how they are used to achieve sharing and isolation of network, compute, storage, and memory resources. Finally, we provide a brief overview of options regarding management and orchestration mechanisms (Section II-C) alongside their compatibility with the aforementioned virtualization mechanisms.

A. Set of Network Functions and Building Blocks

As outlined above, there are numerous different VNF types and areas of use. Specialized surveys and articles provide information regarding individual functions and their details [1], [2], [4], [5]. In contrast, we first (1) identify common building blocks that can be used to compose and classify a particular function. We then (2) demonstrate how this classification and decomposition can be applied to the Next Generation Firewall (NGFW) which features a large number of these building blocks, and later (3) present guidelines regarding acceleration options for the NGFW.

1) Individual functions and blocks: Each building block is characterized by a certain degree of resource utilization (e.g., compute, memory, and storage) as well as the position within a function at which the corresponding action is usually performed. For several VNFs, Table I displays their decomposition into building blocks alongside the each blocks' characteristics, which are outlined next.

Header read. Basically all network functions that are considered in this work need to parse packet headers as a part of their processing and decision making process. Headers are typically compact and well defined, often in binary or Type Length Value (TLV) format, for which this operation is neither compute- nor memory-intensive and requires no access to storage.

Header write. In order to perform the tasks of a Network Address Translation (NAT) function or enable tunneling for Virtual Private Network (VPN), or even for simple IP forwarding, NFs also need to modify the content of packet headers. Although the complexity of such a write access depends on the particular type of operation and incurs a larger CPU and memory usage when compared to a read-only header access, it still amounts to a negligible factor on modern systems.

Table lookup. Often, to make decisions regarding network actions (e.g., like the appropriate forwarding of an IP packet or an Ethernet frame), a lookup based on some packet properties (e.g., the IP or Ethernet destination address) in a specific table (e.g., typically implemented as tries in the case of IP forwarding, or as hash-based data structures for Ethernet)

is necessary. These tables are typically kept in memory, so that no additional storage is required. Hence, CPU usage and memory I/O are higher, particularly in the case of larger forwarding tables.

Payload inspection. More sophisticated actions such as Deep Packet Inspection (DPI) that are performed by an Intrusion Detection System (IDS) such as Bro [6] or Snort [7] or by an NGFW, take into account packet payload to identify malicious content and applications. Depending on the complexity of the performed analysis, this block can be both highly compute- and memory-intensive.

Payload modification. Similarly to the case of header read and write operations, payload modification requires additional CPU cycles for copying packet contents to memory. Furthermore, functions like TCP optimizers and VPNs perform compute-intensive tasks related to compression and encryption, respectively.

Flow state tracking. Although they perform actions on a per-packet basis, several functions need to maintain per-flow state information. This information can range from simple header characteristics in a NAT lookup-table, to data that is more demanding in terms of CPU and memory resources like a short-term history of a flow's packets for tasks like monitoring and intrusion detection [6], [8] or resource sharing for concurrent flows [9].

Flow actions. Furthermore, functions such as firewalls and load balancers perform actions on a per-flow basis. These include flow termination or redirection, are usually performed at the end of a function, and consume few resources.

Session management. Several functions like Intrusion Prevention Systems (IPS), need information on multiple flows in order to reconstruct and analyze the corresponding application sessions and persistently store data that is relevant for accounting purposes. These tasks can incur an even higher demand for CPU and memory resources than their per-flow counterpart. Similarly to the case of flow state tracking, some information might be logged to disk.

Disk access. A subset of NFs heavily depends on read and write access to persistent storage (e.g. nDPI [10]). These NFs include caches as well as network probes that capture and log traffic traces, at either packet or flow-level.

Signal processing. Finally, CPU-intensive signal processing tasks like the Fast Fourier Transform (FFT) need to be performed by Cloud Radio Access Network (C-RAN) functions such as the Base Band Unit (BBU) in a virtualized Base Station (BS).

As mentioned above, building blocks can not only be mapped to the consumption of a particular resource but also to a position within a function. This factor is relevant when considering to accelerate a building block since its position can affect the feasibility and performance gain of the acceleration. While we acknowledge that the position of a building block can vary from function to function, we observe certain patterns, e.g., reading headers and payloads being most likely performed at the beginning of a processing pipeline whereas flow-level actions are usually performed at the end.

2) Next Generation Firewall: In order to illustrate the characterization of a network function by means of its building

Table I: Network functions, the building blocks they are composed of, and characteristics of the building blocks w.r.t. their resource usage. Acronyms: Base Station (BS); Baseband Unit (BBU); Deep Packet Inspection (DPI); Intrusion Prevention System (IPS); Firewall (FW); HTTP Header Enrichment (HTTP HE); Intrusion Detection System (IDS); Network Address Port Translation (NAPT); Next Generation Firewall (NGFW); TCP Optimizer (TCP Opt); Virtual Private Network (VPN).

Network Function	Building Blocks									
	Headers		Table	Payload		Flows		Session	Misc	
	(1) Header Read	(2) Header Write	(3) Table Lookup	(4) Payload Inspection	(5) Payload Mod.	(6) Flow State Tracking	(7) Flow Actions	(8) Session Mgmt.	(9) Disk Access	(10) Signal Proc.
BS / BBU										✓
Cache	✓	✓	✓	✓			✓		✓	
DPI, IPS	✓		✓	✓		✓	✓	✓		
FW, Load Balancer	✓		✓			✓	✓			
HTTP HE	✓	✓	✓			✓		✓		
IDS	✓		✓	✓		✓		✓		
Monitor	✓		✓	✓		✓		✓	✓	
NA(P)T	✓	✓	✓			✓				
NGFW	✓	✓	✓	✓	✓	✓	✓	✓		
Proxy	✓	✓	✓	✓		✓		✓		
Shaper	✓	✓	✓			✓	✓			
TCP Opt, VPN	✓	✓	✓	✓	✓	✓	✓			
Transcoder	✓		✓	✓	✓	✓			✓	
Characteristic	(1) Header Read	(2) Header Write	(3) Table Lookup	(4) Payload Inspection	(5) Payload Mod.	(6) Flow State Tracking	(7) Flow Actions	(8) Session Mgmt.	(9) Disk Access	(10) Signal Proc.
Compute	ε	ε	+	+ / ++	++	+ / ++	ε	++	+	++
Memory	ε	ε	+	+ / ++	+ / ++	+ / ++	ε	++	+	+ / ++
Storage	-	-	-	-	-	+	-	+	++	-
Position	Start	End	Start	Start	Middle	Middle	End	Middle	Middle	Middle

“-”, “ε”, “+”, and “++” represent no, low, medium, and high usage of the corresponding resource, respectively.

blocks, we consider the Next Generation Firewall (NGFW) that is outlined in [11]. In addition to standard firewall features, the NGFW is also capable of performing DPI and encompasses IPS as well as Secure Socket Layer (SSL) VPN functions and Man-in-The-Middle (MITM) proxying capabilities. We deliberately choose this VNF due its wide range of functionalities and the resulting complexity of characterizing it in its entirety. From the capabilities of the NGFW, we can identify its building blocks and requirements: after reading the header of a received packet (1), the NGFW needs to check the packet’s flow and / or session membership (3, 6, 8). Furthermore, a scan of the packet’s payload might be necessary (4) before deciding whether to forward, redirect, or drop the packet (7). Finally, packet headers need to be rewritten (2), sometimes also with payload modification (5) e.g., in the case of proxying or for encryption in the case of VPN tunneling.

B. Virtualization Techniques

In order to fully reap the flexibility and scalability benefits of the NFV paradigm, the underlying infrastructure needs to expose access to a shared resource pool. Simultaneously, it needs to maintain isolation between different tenants and

dynamically adapt the resource allocation to VNF instances for scalability and efficiency. In this context, conventional host resources such as compute, memory, and storage are virtualized alongside networking resources, i.e., links between nodes with delay and bandwidth guarantees. In the following, we provide an overview of virtualization mechanisms from these two categories and refer to them as (1) hardware and (2) network virtualization, respectively. The choice of options from these two categories results in different trade-offs regarding costs, performance, and flexibility.

1) *Hardware Virtualization*.: The main goal in the context of hardware virtualization consist of providing a functional equivalent of the original hardware to the VNF while limiting performance degradation. This is achieved by the virtualization layer that resides between the physical and virtualized infrastructure. Options regarding its implementation range from different hypervisors to container-based approaches which offer different degrees of resource and isolation, security, and performance [2], [12].

Hypervisors can reside at different locations in the system. On the one hand, hypervisors such as Xen [13] and ESXi [14] run directly on the physical machine and directly handle hardware access as well as interrupts. On the other hand,

hypervisors like KVM [15] require the presence of kernel modules in the host OS and kernel processes are responsible for handling events and interrupts. Due to the different numbers of intermediate abstraction layers, implementation details, and the heterogeneity of possible use cases, there is no single hypervisor that performs best in all scenarios and therefore should be chosen carefully to match the particular requirements [16]. Furthermore, high-speed I/O virtualization plays a crucial role for meeting performance requirements and maximizing flexibility when VNFs are run inside VMs [17]. To this end, drivers such as virtio [18] and ptnt [19] provide a common device model for accessing virtual interfaces.

Recently, *container-based* technologies like Docker [20] have gained popularity, that instead share the kernel of the host OS and provide isolation through segregated process namespace (anciently known as BSD jails [21]). On the one hand, containers avoid the overhead of providing an entire guest OS, enabling a flexible and lightweight deployment. In particular, network ports or entire interfaces can be directly assigned to a container without incurring an overhead as in the case of hypervisor-based solutions. On the other hand, sharing the kernel also raises concerns regarding security and (performance) isolation between instances. Finally, worth mentioning are also *unikernel* [22] as well as *serverless* [23], [24] technologies for the virtualization of network functions, that sits at the opposite side of the spectrum as the former targets full flexibility with light-weight monolithic kernels that run on a hypervisor such as Xen, whereas the latter targets performance by trading it for a more constrained environment offering a set of Application Program Interface (APIs). All these technologies are still competing, and there is no single candidate that fits all use-cases, which makes the ecosystem very rich.

2) *Network Virtualization*.: In a similar fashion to hardware virtualization, network virtualization (NV) accomplishes an abstraction of network resources and therefore acts as an enabler for fully programmable networks whose entire stack can be provisioned in a flexible manner. The survey in [25] provides an overview of NV approaches that are categorized w.r.t. several characteristics. These include the network technology that is virtualized, the layer in the network stack at which virtualization is performed, as well as the granularity of virtualized entities. Depending on the envisioned use case, operators need to decide on an NV technology that satisfies the corresponding requirements.

An additional step in the evolution of NV consists of combining the NV idea of sharing a physical substrate among several stakeholders with the SDN paradigm. By introducing an SDN hypervisor, it is possible to virtualize SDN-based networks and therefore allow multiple tenants to deploy their own SDN-based network including their own controllers while operating on one common physical infrastructure. A survey of SDN hypervisors is available in [26]. The main distinctions between different of these hypervisors include their architecture, the network attributes that are abstracted, as well as the capabilities in terms of isolation between slices.

Our work leverages the technologies described in [26] and clearly build upon [25]: specifically, we go one step further

with respect to [25] by understanding how functions can be implemented and, especially, accelerated with either software, hardware or hybrid approaches.

C. Integration into the Orchestration Context

Given the abstractions and flexibility of the outlined virtualization mechanisms, management and orchestration (MANO) techniques can attach to interfaces that are defined in frameworks such as ETSI MANO [27] and adapt the resource allocation of individual VNF types and instances in a dynamic fashion. Additionally, knowledge regarding a VNF's resource requirements and performance levels given a specific resource enables scaling VNFs according to observed demands. An overview of efforts from research projects, standardization bodies, and both open-source and commercial implementations is provided in [28].

A concrete example regarding the integration of hardware acceleration mechanisms into the MANO context is outlined in [11]. In the presented use case, a Virtual Infrastructure Manager (VIM) enables the elastic deployment of an IPsec tunnel termination VNF by combining two types of information. On the one hand, it has knowledge of nodes' CPU cores, available instruction sets, and hardware acceleration capabilities. On the other hand, the VNF descriptor provides the performance levels in terms of possible bandwidth given different numbers of CPU cores, instruction sets, and hardware acceleration.

Whereas orchestration is out of the scope of this survey, [11], [28] testify that effort is ongoing into making it possible to harness acceleration techniques in a seamless way. It is thus important to understand which of the software or hardware acceleration are suitable for VNFs, and more generally to outline a pattern for the design and implementation of new VNFs that can make the most out of the available acceleration techniques.

III. SOFTWARE ACCELERATION TECHNIQUES

To bridge the performance gap between specialized hardware middleboxes and software instances that run on COTS hardware while retaining the flexibility of the latter, the past two decades have seen tremendous advances in software acceleration techniques (Section III-A). This sparked a quick evolution of pure-software user-space network stacks and NFV frameworks, which attain multi-gigabit performance on COTS platforms (Section III-B).

A. Software Acceleration

We review the most important software acceleration techniques in the context of network function virtualization. In the below discussion we make a distinction between *pure software* acceleration techniques, program code optimization approaches affecting the parts of the network stack, including firmware, driver code, operating system code, and the user-facing network infrastructure, that can be directly modified by a user or a vendor without having to upgrade or modify the underlying hardware, and *hardware-assisted acceleration techniques* that require explicit support in hardware. Table II provides a compact summary of the techniques and their benefits.

Table II: Benefits of software acceleration techniques. Acronyms: Zero-copy (ZC); Mempools (MP); Hugepages (HP); Prefetching (PF); Cache alignment (CA); Lock-free multithreading (LFMT); Lightweight threads (LT); Multiloop (ML); Branch prediction (BP); Receive-side scaling (RSS); Flow hash (FH); Single-Instruction Multiple-Data (SIMD); Direct Data I/O (DDIO); Single-root input/output virtualization (SR-IOV).

	Poll	I/O Batch	Memory					Compute Batch	Threading		Coding		NIC-support			CPU-support	
			ZC	MP	HP	PF	CA		LFMT	LT	ML	BP	RSS	FH	SR-IOV	SIMD	DDIO
Reduce memory access			✓		✓	✓	✓										✓
Optimize memory allocation				✓	✓			✓									
Share overhead of processing									✓			✓					
Reduce interrupt pressure	✓	✓															
Horizontal scaling										✓	✓		✓		✓		
Exploit CPU cache locality								✓	✓	✓							✓
Reduce CPU context switches	✓	✓								✓	✓						
Fill CPU pipeline									✓			✓	✓				✓
Exploit HW computation														✓	✓	✓	✓
Simplify thread scheduling	✓									✓							

1) *Pure Software Acceleration:* Polling techniques, as opposed to an interrupt-driven design, have been used for quite some time to speed up Network Interface Card (NIC) drivers. In the interrupt-driven mode, the NIC driver is managed by the operating system in the kernel-space or a user-space application via kernel-bypass, and is responsible for pushing received packets to main memory and to interrupt the CPU to signal the availability of new packets to process. The CPU then enters a special interrupt-handling routine (IRQ) and starts VNF processing. However, the corresponding context switches may cause non-negligible overhead at high loads and the interrupts may saturate the CPU. *Interrupt-mitigation* techniques have a relatively long history: one such technique is represented by *interrupt-coalescing* [29] which waits for a fixed timer for further packet reception before raising an interrupt. Yet, a more efficient technique to avoid overloading the CPU at very high packet rates is switching to *polling mode*, whereby the CPU periodically checks for pending packets (pull model) without the need for the NIC to raise an interrupt (push model). Polling the NIC in a tight loop, however, results in 100% CPU usage regardless of the traffic load. Support for poll-mode drivers has been introduced in BSD at least since 2001 [30] and interrupt coalescing is also available in the Linux kernel [31].

I/O batching is another technique that is commonly used in poll-mode NIC drivers to mitigate interrupt pressure at high load. In batched mode, the NIC does not receive or transmit packets individually, but operates by aggregating several packets into a batch. Upon reception of a packet, the NIC writes it into a hardware queue and then the whole batch is pulled by

the CPU in a single chunk. In the reverse direction, the CPU may wait for multiple packets to accumulate in the transmit queue before performing an I/O operation. Accordingly, I/O batching overcomes the bottleneck that exists between the NIC and the main memory by amortizing I/O costs over multiple packets. The price to pay is an increased per-packet latency, due to packets having to wait in the queues until a complete batch is formed. First used in PacketShader [32] and DoubleClick [33], I/O batching has become widely adopted by all major high-speed packet processing frameworks to today [34]–[36].

Optimizing memory management is another technique that may significantly improve the performance of VNFs. Upon finalizing packet reception, the kernel needs to make the received data available for a user-space application to consume it. This operation may be an additional bottleneck in traditional network stacks, since the memory region where packets can be written by the NIC via Direct-memory Access (DMA) may not overlap with the user-space memory, incurring a memory copy overhead. Several techniques exist to overcome this limitation. A *zero-copy* network receive/transmit code-path [35]–[37] mitigates costly memory operations by mapping DMA regions into user-space memory and passing only lightweight packet descriptors between the kernel and the application; such descriptors may contain a pointer to the packet buffer plus some additional metadata.

Memory management, i.e., the allocation of new packet buffers or the freeing of unused ones, may also be a source of significant overhead. To avoid memory management in the fast packet processing path, modern network stacks typically

use pre-allocated packet buffers. A set of memory pools is created during startup and this memory area is never freed: such *mempools* are usually organized into ring buffers where packets are written to, and read from, in a sequential fashion. Mempools are usually allocated from *hugepages* whose size typically ranges from 2 MB to 1 GB to prevent misses in the Translation Lookaside Buffer (TLB). Care must be taken, however, to place mempools to the *Non-Uniform Memory Access* (NUMA) node where the majority of memory accesses will be made during runtime, otherwise performance may incur the penalty of accessing faraway memory across NUMA nodes. Network programmers should also take care of adopting *cache-friendly* data structures: for instance hash-table buckets should always be sufficiently aligned to cache lines and should occupy as few cache lines as possible in order to maximize CPU cache hit rate, and *compressed data structures* may be used to reduce the overall memory footprint of performance-sensitive data [38]. Finally, *prefetching* data ahead of time from main memory may substantially contribute to increased performance, by avoiding CPU stalls (i.e., that the execution of CPU instructions is blocked due to memory access).

Compute batching is a technique that can be used to enhance the CPU pipeline performance [34], [39]. Traditional network engines typically process network traffic on a per-packet basis: network functions in the processing path are sequentially applied to a packet until the final forwarding decision is made. With compute batching, the notion of I/O batching is extended to VNF processing, in that network functions are implemented from the outset to work on entire bursts of packets rather than on a single packet. Similarly to I/O batching, compute-batching helps mitigating the overhead of invoking VNFs (e.g., context switches and stack initialization) as well as providing additional computational benefits. First, compute batching optimizes the use of the first-level CPU instruction cache: when a packet batch enters a network function, the corresponding code is fetched into the L1 instruction cache upon processing the first packet, and the instruction cache remains warm during processing the rest of the packet burst. Furthermore, using compute batching allows to take advantage of data-level parallelism primitives available in modern CPUs (see SIMD later). Finally, compute batching also allows better use of the CPU processing pipeline, as it facilitates the CPU pipeline to be consistently full, which allows the CPU to execute multiple instructions within the same clock cycle [34].

Exploiting CPU parallelism has become imperative given that, to manage CPU power dissipation, processor makers started favoring multi-core chip designs. Thus, network applications must be written to exploit a massively *multi-threaded* design. Packet processing presents a workload that is ideally suited for parallelization, in that each packet or packet batch may be serviced by a separate execution thread, but care must be taken to avoid packet reordering, and especially to preserve flow-coherence (in both the forward and backward flow directions). To maximize multicore scalability, it is imperative to *avoid synchronization* between threads as much as possible, e.g., by preferring lightweight mutual exclusion primitives like read-copy-update (RCU) over mutexes and semaphores, and leveraging *atomic instructions*, *lock-free* data

structures, and *per-CPU* variables. Lock-free parallelism is tightly coupled with the availability of multiple hardware queues, since different CPUs can be attached to different hardware queues (see RSS later) to operate on different traffic subsets. More recently, frameworks such as DPDK [35] started complementing the fully-fledged preemptive multitasking offered by the underlaying OS, with the introduction of a simpler *cooperative multitasking* model. With standard *run-to-completion execution*, each packet is assigned to a separate thread that performs a complete VNF processing until the final forwarding decision. In a cooperative environment, a thread can implement a subset of functionalities, and can decide when to pass the packet to another thread for further processing. This was made possible by the introduction of lightweight threads, making the VNF scheduler logic much simpler and faster.

Programming best-practices also becomes imperative for network application developers, as some coding techniques are known to help compilers to accelerate packet processing. One such best-practice is called *multi-loop* programming, an evolution of the conventional *loop unrolling* practice by explicitly writing the program control flow in a way as to handle packets in groups of two or four in each iteration (dual-loop or quad-loop, respectively). Loop unrolling has two main goals: the first is to reduce the number of "jump" instructions (that can introduce idle slots in the CPU pipelines). The second goal is to carefully prefetch new packets while operating on already prefetched packets: this coding practice helps the compiler to organize the code in a way as to maximize the usage of the CPU pipeline [34]. In addition, explicitly *inlining* frequently accessed code may allow to save on runtime function call overhead, at the cost of a slightly increased program text size [40]. Finally, using workload-related insights a network programmer may manually annotate the branches of conditional program code more likely to execute during runtime, this way improving the CPU's *branch prediction* success rate. Using such programmer-provided annotations, the compiler can place the likely execution branch right after the conditional expression and let the CPU automatically fetch the corresponding code into the CPU pipeline; in contrast, the CPU pipeline must be tediously invalidated and repopulated with the correct branch code in case of any mispredicted branch, possibly leading to a significant performance penalty.

2) *Hardware-supported Functions in Software*: An important class of acceleration techniques is constituted by hardware-assisted functions, whereby the hardware exposes certain functionalities that can be used to speed up the execution of software. We distinguish two categories in this context, depending on whether the assistance is offered by the NIC or by the CPU, that are reported in the rightmost part of Table II.

NIC-assisted acceleration techniques range from virtualization support and direct DMA to NIC-driven parallel packet dispatching. Modern NICs contain a fast packet parser to compute flow level hashes in hardware, maintain multiple hardware packet queues called *Receive Side Scaling* (RSS) and expose packet counters in registers. Access to this hardware functionality is typically implemented in the low-level NIC drivers. Use of register-backed packet counters reduces memory overhead, while RSS is instrumental to multi-threading

process, as the packet RSS hash may be used by the NIC to dispatch packets to different CPU cores, in order to leverage flow-level parallelism and avoid packet reordering. RSS hashes ensure that packets of a single transport-level connection (and, depending on the RSS seed, of both direction of the connection) will always be scheduled to the same CPU, which also enforces locality of data structures usage. Furthermore, newer NICs can take advantage of *Data Direct I/O* (DDIO), which allows packets to be transferred directly into the last-level CPU caches instead of into the main memory, preventing a costly cache miss when the CPU starts processing the packet. *Single root input/output virtualization* (SR-IOV) in addition lets the NIC arbitrate received packets to the correct VNF without the explicit involvement of a hypervisor switch.

CPU-assisted acceleration techniques, on the other hand, leverage the features of modern CPUs to speed up network applications. Most modern CPUs support low-level data-parallelism through an advanced *Single Instruction Multiple Data* (SIMD) instruction set. SIMD operations allow to execute the same instruction on multiple data instances at the same time, which greatly benefits vector-based workloads like batch packet processing. In addition, latest CPU chipsets include built-in *CPU virtualization* support, exposing a single CPU as multiple virtual CPUs to different VNFs, *hyperthreading* to multiply the number of CPU pipelines for better multicore scalability, and multiple *memory channels* that allow applications to load-balance memory accesses.

B. The Ecosystem of Software Stacks

The previous techniques are leveraged by a number of tools, that we overview in this section. Roughly, software stacks can be categorized into three main branches, depending on whether they are (1) low-level building blocks, (2) specialized pieces of software implementing a single VNF, or (3) full-blown frameworks for network function composition. In the following, we discuss representatives from each category and provide a visual overview of the resulting ecosystem of stacks in Figure 2.

1) *Low-level Building Blocks*: Low-level high-speed packet processing libraries may be implemented in kernel space or in user-space, using kernel-bypass. Kernel-space solutions have full access to the resources of the machine, but must be accessed via standard system calls (the kernel performs controls to grant fault tolerance and isolation w.r.t. other processes). User-space approaches, on the contrary, bypass this step via specific libraries, thus avoiding the overhead of the kernel at the cost of a reduced isolation. Particularly, while kernel-based networking has progressed at a relatively low pace, user-space data-plane libraries have seen tremendous advance within the last decade, with a flourishing ecosystem of which [35]–[37] are representative examples. A performance comparison of these frameworks for packet forwarding is available in [39].

Netmap [36] is a framework for packet I/O that provides a high-speed interface between user-space applications and the NIC (with assistance of a kernel-level module). Depending on the NIC utilized, netmap may leverage hardware capabilities (such as RSS queues), and it allows to reach a 10Gbps rate

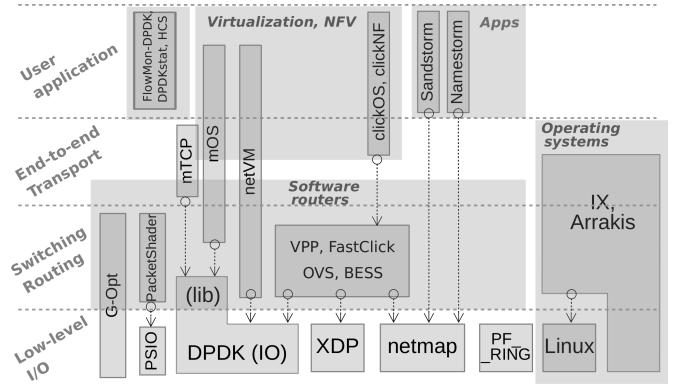


Figure 2: The ecosystem of software stacks for high-speed network function composition.

with a single low-tier CPU (900 MHz CPU clock rate). Being part of FreeBSD and also available on Linux, netmap is well-known in both the research and industrial communities, and is typically used to build packet-sniffing or traffic-generator applications, L2/L3 software routers, and even firewalls. The Intel Data Plane Development Kit (DPDK) [35] is another framework for high-speed packet processing, with a large user community. In contrast to netmap, DPDK provides a full kernel-bypass, exporting the NIC driver logic as well into user-space and exposing a rich NIC abstraction to applications: registers, counters, and hardware-assisted functionality may all be accessed by the programmer using a convenient C language API. PF_RING_ZC [37] is another multi 10-gigabit framework targeting inter-process and inter-VM communications. PF_RING_ZC NIC drivers may work both as a regular kernel driver or can be switched into kernel-bypass mode, allowing the user to select which packets are received in kernel-bypass mode and which ones get processed by the standard Linux kernel network stack.

Recently, the above features used by kernel-bypass stacks have started being introduced in the Linux kernel, notably by AF_PACKET v4 [41] and the eXpress Data Path (XDP) [42] projects. XDP adopts many of the aforementioned acceleration techniques, such as batched I/O, multiple HW queues, and zero-copy, on top of a virtual machine that allows the user to inject custom packet processing programs into the network stack written in the C-like eBPF language. Applications can also take advantage of the MSG_ZEROCOPY feature available in recent Linux kernels that enables zero-copy transmission of data, and efforts are ongoing to address the receive code-path as well [43]. Given the recent appearance of these function in the kernel, a comprehensive comparison of high-speed kernel and kernel-bypass stacks is still lacking so far.

2) *Monolithic Network Functions*: One step above in the ecosystem is represented the realization of useful network functions based on the low-level building blocks: a set of monolithic network functions have started to appear that address one specific use case, like forwarding, network address translation, intrusion-detection, transcoding, etc. These functions are custom stand-alone software components that either (more rarely) re-implement a custom user-space network stack

making use of the acceleration techniques early introduced or (more frequently) build their VNF-specific code on top of an existing low-level building block, like DPDK or netmap.

One such specialized VNF is PacketShader [32], a high-speed packet I/O framework targeting packet processing functionality is implemented such as IPv4/IPv6 forwarding and OpenFlow switching. PacketShader greatly benefits from GPU acceleration. Another tool is G-opt [44], a software router which targets memory access optimization to speed up packet processing. The Hierarchical Content Store (HCS) [45], targets the problem of line-rate access to a router cache in Information Centric Networking [46].

Sandstorm and Namestorm [47] are specialized user-space network stacks building over netmap to implement Web server and DNS server functionalities, respectively. DPDKStat [8] and FlowMon-DPDK [48] are, as the name implies, DPDK-based tools for traffic monitoring: the former is a CPU-hungry tool providing features such as DPI, full payload analysis, flow reconstruction and exports hundreds of advanced flow-level statistics, the latter targets very simple flow-tracking with minimal CPU resources. mTCP [49] is a user-space TCP framework targeting high-speed flow-level packet processing, designed to exploit multicore systems and it can work on top of Linux as well as PacketShader I/O, DPDK, or netmap.

Finally, Open vSwitch (OVS) [50] and ESwitch [40] are examples of fully fledged high-performance OpenFlow software switches that come with some network functions built-in (e.g., NAT with connection tracking). OVS in particular is inspired by the SDN paradigm of the separation between control and data plane, and as such works with a set of match/action rules, similarly to a regular SDN switch, and it can be deployed both over a stock Linux kernel using kernel-space networking and also over a fast DPDK datapath.

3) Modular Network Function Composition Frameworks:

One last relevant class of the software stack ecosystem is represented by frameworks whose main aim is generality and feature-richness: in contrast to low-level blocks or monolithic functions, these frameworks provides a set of already available VNFs, as well as an environment which simplifies the development of new VNF and, especially, their composition into complex and dynamic service chains.

In this category we find software routers, such as Vector Packet Processor (VPP) [34], Click [51] and its high-speed variant FastClick [39], and Berkeley Extensible Software Switch (BESS) [52]. These frameworks allow users to configure, manage, and deploy flexible service chains, by developing and customizing monolithic network functions such as ACL and NAT as elementary building blocks and then arranging these network functions into a dataflow graph. At high-level, the difference is that VPP is aimed at typical L2-L4 workloads and, as such, the dataflow graph is fixed, whereas Click and BESS allow arbitrary processing graphs, thus focusing more on flexibility and reconfigurability.

While the previous frameworks delegate the majority of the work for developing and arranging VNFs to the user, other frameworks offer a “cloud-like” environment for the flexible composition of network functions encapsulated as stand-alone virtual machines or containers. Examples are ClickOS [53],

ClickNF [54], mOS [55], OpenStack [56] and NetVM [57]. ClickOS [53], provides tiny virtual machines (unikernels) that implement the Click abstraction for providing resource-efficient and high-performance virtual machines. ClickNF [54] and mOS [55] respectively build over Click and mTCP, to offers the possibility to design, prototype, and deploy custom middleboxes on commodity hardware. OpenStack [56] and NetVM [57] in addition, provide telco cloud services to deploy customizable data-plane functionality as virtual machines, targeting a high-speed inter-VM communication paradigm.

Finally, worth mentioning are IX [58] and Arrakis [59]. Rather than targeting composition of VNF through a dataflow graph abstraction or in a cloud-like NFV infrastructure, these frameworks offer a new operating system that is explicitly tailored for network function deployment. In particular, both IX and Arrakis re-implement basic operating system functionality, including I/O, memory management, and system calls, to provide a protected environment in which high-speed network functions can be deployed as simple network applications.

Given the abundance of tools for VNF deployment with similar spirit and capabilities, making a selection of the most appropriate candidate is not an easy task. Particularly, very few work exists that aim at independently and directly benchmarking these frameworks in a scientifically sound way. Further work beyond [60], that compares OVS, SR-IOV and VPP, is thus necessary to provide a complete picture of the ecosystem.

IV. HARDWARE OFFLOADING

With the end of Dennard’s scaling and Moore’s law, general purpose CPUs cannot increase their processing power at the same pace at which per-interface network speeds grow [61]. As a matter of fact, Network Interface Cards (NICs) already offload several functions that should be otherwise provided by a system’s CPUs. For example, commodity NICs such as those described in [62], [63] can offload checksum calculation and verification, packet filtering, and functions related to specific protocols such as VxLAN encapsulation or TCP segmentation. Such operations help the system in handling traffic rates in excess of 40 Gbps per NIC port, saving the CPU for further computations.

In line with this trend, and in order to handle a larger number of potentially complex functions, the adoption of *programmable* network function accelerators is becoming widespread¹. Generally, these accelerators are integrated in the NIC, hence called a SmartNIC [65]–[67], and deployed as *bump-in-the-wire*, i.e., they apply their processing either before or after the processing performed by the system’s CPUs.

Current SmartNICs’ programmable engines are usually provided with proprietary software that offloads some well-known protocols and functions, and whose programmability is not exposed to third-party users. This allows to quickly adapt the SmartNIC functions to emerging needs, without waiting for the long implementation cycles experienced with traditional

¹Such a need has been recently recognized also by standard organizations such as ETSI ISG NFV, which is defining hardware accelerators’ capabilities in their specifications for NFV platforms [64].

NIC hardware [68], but it is a process still largely driven by specific vendor's interests. For example, it is common for SmartNICs to provide only well-known functions, such as Firewall offloading, e.g., integrated with Linux's iptables, or offloading of virtual switches, e.g., Open vSwitch [50].

However, the offloading of *fixed* functions hardly matches the requirements of modern dynamic and fast changing applications [69]–[72]. As a result, recently, vendors started to increasingly open their programmable engines, and provide users with the ability to customize the SmartNIC's software. As such, it becomes of particular interest to study how SmartNICs can be leveraged to become programmable network function accelerators, and what are the related opportunities and issues.

A SmartNIC [73] typically adopts specialized Network Processors (NPs) or FPGAs as programmable engines. Despite obvious considerations regarding costs, with the NICs being usually cheaper than SmartNICs, the latter have also important implications in terms of ease of use, application development time, and performance guarantees, to name a few. In particular, while a SmartNIC gives great flexibility to introduce new functions enhancing system performance, it comes with the complexity of programming their engines, which includes phases of design, debugging, verification, and optimization. To overcome this design complexity, SmartNICs often offer higher level, domain-specific programming abstractions for their programmable engines [74], [75]. Such abstractions implement a point in a conceptual design space that ranges from fixed functions on the one side, to full programmability on the other, with any specific solution trading in its way programming simplicity with expressiveness.

In this sense, one of the open research questions for SmartNICs is about which abstraction should be adopted, in order to provide ease of use, flexibility, and performance at the same time. In the remainder of this section, we first describe the primitives usually offloaded to commodity NICs (Section IV-A), then we give a short survey of the SmartNIC architectures (Section IV-B). Finally, we describe the currently available programming abstractions for SmartNICs' offloading engines (Section IV-C).

Here, we want to underline that there are complex tasks, such as Deep Packet Inspection (DPI), which are appealing candidates for hardware acceleration but that are not covered in this survey. For example, Deep Packet Inspection (DPI) is a candidate for hardware acceleration due to the cost of payload inspection and to the complexity of pattern matching operations. In fact, DPI acceleration has been extensively considered both using different targets like GPU [76] and FPGA [77] and using different algorithmic approaches [78]–[80]. However, due to the specific nature and complexity of such functions, the corresponding acceleration techniques require an ad-hoc and extensive description that is outside the scope of this survey. We instead refer the interested reader to [81] for further details.

A. Offloading in Commodity NICs

Here we describe a set of commonly used primitives offloaded to NICs and we discuss to what extent these offloading

procedures can be made programmable in order to satisfy the flexibility required by NFV applications. A set of widely offloaded primitives is listed next: clearly, commodity NICs offer a set of well defined fixed functions targeting existing widespread protocols.

1) *CRC/Checksum Computation*: The widespread use of some checksum computation such as the 1's complements sum used by TCP/IP protocols makes it suitable for hardware offload. In transmission, the NIC directly computes the checksum value defined by the specific protocol and sets the corresponding field in the packet before sending it out; in receive mode, the NIC checks the checksum's correctness and discards the packet if the checksum is wrong. Almost all commercial NICs provide TCP/IP checksum offloading. Since the Stream Control Transport Protocol (SCTP) checksum algorithm is different from TCP/IP, only few NICs support SCTP checksum offloading. This is clearly related to the smaller adoption of this protocol with respect to TCP. It is worth noting that checksum computation could be easily performed by current generation processors, also exploiting specific instruction set extensions to speed up the computation. This limits the gain between hardware and software implementations. However, the cost of implementing such operations in hardware is marginal, thus making its offloading appealing. Furthermore, offloading the checksum enables also other offloading features such as TCP Segmentation Offloading or IPsec offloading. Finally, even if the CRC/checksum computation is fixed and not programmable in current NICs, it seems not necessary to develop more flexible and programmable engines for checksum computation due to the very limited interest in deploying checksum algorithms different from those used today.

2) *Encryption/Decryption*: The encryption/decryption of packet payloads requires very expensive computations, and as such it a good candidate for NIC offloading. Due to the large number of encryption standards, each one with its specific algorithm and implementation characteristics, it is nearly impossible² to have a generic or programmable encryption offloading engine. Instead, there are several NICs that are able to provide specific protocol offloading, such as IPsec, with a subset of encryption algorithms (e.g., AES-CBC) with authentication features (e.g., HMAC-SHA1 and HMAC-SHA2 with different key length), or more recent AEAD - Authenticated Encryption with Associated Data - algorithms such as AES-GCM or AES-CCM. In some cases, the encryption offloading can save more than 50% of CPU time, providing significant gains of the packet forwarding throughput [82]. However, due to the complexity of the hardware development of encryption algorithms, only high end NICs provide IPsec offloading features. Moreover, integrating the encryption engine in the ASIC chip providing the basic NIC functionalities can be very costly. A possible solution utilized by several vendors is to use a Network Processing Unit (NPU) or an FPGA inside the NIC as the device in which the encryption engine is realized. This choice presents several

²Nor advisable! As well known, implementation of cryptographic primitives requires professional developers with extremely specific skills, e.g. to avoid secret information leakage via side channels that a general-purpose developer may inadvertently introduce in the design.

benefits, since it is possible to add novel algorithms updating the NIC firmware, and it makes very easy to provide patches if security problems arise in the implementation of the specific protocol. From the programmability point of view, this is an opportunity for the development of NFV, since the presence of a programmable device in the NIC could permit a more timely network security upgrades and a much faster patching of emerging vulnerabilities. In short, the hardware designed to provide programmable encryption engine already represents an embryo of an FPGA enabled SmartNIC, as those described next.

3) *Large Send/Receive Offloading*: Transport protocols usually split the stream of data to transmit into smaller chunks with a maximum size given by the network infrastructure. The task of splitting a stream in smaller segments, as well as the receiver task of rebuilding the stream from the received packet can consume several CPU clocks. In particular, there is a considerable overhead caused by the system call associated with each packet. These overheads can be avoided if the task of splitting / rebuilding the packets is offloaded to the NIC. In case of data sending, this is generically called generic segmentation offload (GSO) and it is called TCP segmentation offload (TSO) when applied to TCP. The offloading engine breaks the data coming from the host into smaller segments, adds the TCP, IP, and data link layer protocol headers to each segment, recomputes the checksum, and sends the packet over the network. For the receive side, the offloading is called Large Receive Offload (LRO) and allows the system to reduce the CPU time for processing TCP packets that arrive from the network, and to reduce the interrupt pressure from the NIC. We notice that these techniques complement the I/O batching technique early seen: whereas I/O batching maintains the individual packets received by the card unaltered and reduces interrupt pressure, segmentation offloading goes one step further by offering to disassemble/reassemble higher-layer data into/from several segments, which reduces interrupt pressure. Furthermore, LRO also can significantly increases the throughput of software network functions. In fact, in software NFs most of the processing overhead comes from per-packet operations. Increasing the packet size well above the MTU size (a reassembled packet could have up to packet 64KBs) drastically reduces the overhead thereby increasing the overall throughput of a software network function.

Still, it is worth to remark that when, as in the case of TCP, packet batching occurs at the NIC level, the subsequent transmission of batched packets may occur in (micro) bursts which may increase packet loss in some scenarios [83]. Programmable pacing techniques directly implemented in the NIC HW are an interesting research direction to address such issue [84].

Another hot topic for segment offloading is related to the use of this technique when UDP is used to build an overlay protocol, for example for QUIC. Since each QUIC packet in its encrypted header a packet number (PN) identifier, the segmentation/reassembly of UDP packets breaks the correspondence between the UDP packet and the PN. And due to the encryption, it is not straightforward to update the PN during the segmentation/reassembly operations. Thus, a native QUIC

segmentation offloading can be done combining both the HW encryption with a modified HW for segmentation offloading. However, as will be discussed extensively in section V-C, designing such offloading hardware depends on how largely adopted will be this protocol.

B. SmartNIC Architectures

A SmartNIC hosts a programmable engine that allows for the implementation and extension of the NIC's functions, e.g., for hardware offloading of new protocols that become of widespread adoption. Almost all the SmartNICs work as a *bump-in-the-wire* device inserted between the PHY Ethernet interface and the host PCIe interface [85]. This type of SmartNIC usually provides several network interfaces, a PCIe interface to communicate with the host, and different memory peripherals such as external DRAMs and SRAMs. The former are bigger memories with considerable latency and are used for packet buffering, the latter has lower latency and are used to store random access information (e.g., hash or routing tables, counters, etc.). The core of the SmartNIC is a programmable element that can be a Multicore System-on-Chip (SoC) or an Field Programmable Gate Array (FPGA), as discussed next.

1) *Multicore SoC-based SmartNICs*: Examples of Multicore SoCs are the Cavium [86] and BlueField [87], based on general purpose CPU cores (mainly ARM cores) or the Netronome [88] and Tilera [89], based on smaller cores specifically designed for networking applications. Usually these SoCs also have several hardware blocks, usually called acceleration engines (AE) that perform specific network tasks such as encryption or packet classification (using on-chip TCAMs). These processor-based SmartNICs also use software acceleration techniques like those described in the previous section (some of them can run applications on top of the DPDK framework) but at the same time share the same drawbacks, in terms of latency and latency variability as well as memory contention, state synchronization and load balancing issues among cores. In summary, these devices allow for a good degree of programmability but offer limited scalability and maximum achievable throughput, e.g., it is usually hard to scale their performance beyond 50-100 GbE [90].

2) *FPGA-Based SmartNICs*: The other solution for the realization of a SmartNIC is a FPGA-based card. A well known example of this kind of SmartNIC, in the academic domain, is the NetFPGA SUME [67], that provides 4x10 GbE SFP+ links, a x8 Gen3 PCIe interface, a Xilinx Virtex-7 XC7V690T FPGA, three 72 Mbits SRAM chips and two 4 GB DDR3 DRAMs. Several other cards with similar characteristics are available in the market. These cards differ for the FPGA performance and size and for the type of attached peripheral (number and speed of network links, size of external memories and so on). The performance of FPGA-based SmartNICs is extremely competitive in terms of latency, power consumption, and cost as extensively discussed in [90].

C. Programming Abstractions

SmartNICs differ not only in their hardware architecture, but also in the way the hardware features can be accessed, pro-

grammed and extended. In the most general case, an FPGA-based SmartNIC can be programmed with Hardware Description Languages (such as VHDL and Verilog), whereas Multi-SoC-based SmartNICs offer programmability through some high-level language (e.g., a C dialect used to programmed specialized network processors). However, since these languages require a programmer to have extensive hardware knowledge (either for FPGA or Network Processors), domain-specific abstractions for network functions are often also provided. We can categorize these abstractions in *stateless* and *stateful* ones. A stateful abstraction allows a programmer to describe algorithms that read and modify state, whereas a stateless abstraction can only read state that has been provided to the device through some other means, e.g., written by a non-programmable block.

1) *Stateless Abstractions*: The match-action table (MAT) abstraction is widely used to describe a stateless packet processing data plane. The abstraction allows a programmer to define which actions should be applied to the packets belonging to a given network flow. The flow is specified by the match part of MAT’s entry, while the corresponding actions can forward or drop a packet and modify its header. The entries of the MAT cannot be changed using the abstraction itself. It is assumed that a separated control plane asynchronously updates the entries. In other words, the entries’ update operations are not executed in the device’s fast path.

MATs are used in several contexts. OpenFlow [91] adopts a pipeline of MATs to describe packet processing in network switches. Some SmartNICs implement OpenFlow-like abstractions, and even regular NICs may offer a limited MAT abstraction support [92]. Reconfigurable Match Tables (RMT) [93] extends the OpenFlow approach with the ability to define the type and number of MATs, including the description of custom packet headers and actions. Originally targeted to hardware switches (P4 switches may indeed leverage an internal architecture based on the RMT technology), the approach has been recently applied also to SmartNICs [69], [74], [75]. Given their stateless nature, MAT-based abstractions are generally used only to describe a small subset of functional blocks, such as Header Read and Write and a small set of Flow Actions.

2) *Stateful Abstractions*: While MAT abstractions dominate the scene for stateless packet processing, it is still unclear which abstraction is most appropriate for capturing high-performance stateful processing. Often a very generic approach is used: the programmer writes tiny *packet programs* that are executed for each processed packet. The programs are typically expressed with a restricted programming language, in order to reduce the risk of negatively impacting performance (e.g., removing unbounded loops), and to increase system safety (e.g., forbidding flexible, pointer-based memory accesses).

This is the approach used by packetC [94], protocol oblivious forwarding [95] (POF), Linux’s enhanced Berkeley Packet Filter [96] (eBPF) and Domino [97], which mainly differ for the target platforms and for the state access model. For instance, packetC and POF target network processors, eBPF is usually executed within the Linux kernel but can be offloaded

to some SmartNICs, while Domino targets high-performance ASIC implementations. In terms of state management, these abstractions are influenced by the target executor. Thus, when the target is a multi-processor system (NPs, multi-core CPUs), state is partitioned in processor-local and global parts, and the programmer has to carefully manage state access. In the case of Domino, the state is always global, and state consistency and performance are guaranteed at compile time at the cost of limited flexibility.

A different, more structured, abstraction was originally suggested by OpenState [98] and FAST [99]. In both cases packets are grouped in programmer defined network flows (like in MATs), and flows are associated with a Finite State Machine (FSM). Programmers define the FSMs transition tables, which are used to update the corresponding FSMs’ states as packets are processed. When processing a packet, the corresponding FSM’s current state is used to select the packet forwarding and modification actions. The compelling nature of such abstractions is their very efficient implementation in hardware, in a small and deterministic number of clock cycles - a state transition in fact can be modelled and executed via a suitably extended flow table [98]. Moreover, OpenState was extended in [100] with the support for triggering conditions, flow context memory registers, and on-the-fly arithmetic/logic register update functions. These stateful abstractions allow a programmer to describe more complex functions, thus supporting also the offloading of more demanding Flow State Tracking and, in some cases, Session Management blocks.

At last, please note that all the stateful HW programming abstractions mentioned above have been designed for the somewhat “simpler” context of programmable data planes inside network switches, i.e., on-the-fly forwarding and packet-level processing functions. And even in such scenario, to date it is still unclear what is the “right” compromise between flexibility, performance, and openness. To a greater extent, such compromise becomes critical when we seek programming abstraction focused on higher layer network functions, which may involve network functions well beyond packet-level “on-the-fly” operations — hence further accounting for protocol states, timers, opportunistic packet buffering, signalling messages, etc. The quest for a sufficiently expressive programming abstraction, tailored to permit offloading in hardware NICs of transport/application layer functions, or even implement way more challenging cloud-like functions such as those posited in [101], is still open and appears to be an exciting research topic.

V. KEY DERIVATIONS AND GUIDELINES

In this section, we present guidelines concerning the different techniques that were detailed in the previous sections. They include general guidelines and caveats that should be taken into account when designing and developing VNFs (Section V-A). Furthermore, we provide guidelines that are specific to the two main categories of acceleration techniques that are covered in this work, i.e., software (Section V-B) and hardware (Section V-C) acceleration. Finally, we demonstrate how these guidelines can be applied to the exemplary NGFW function (Section V-D).

A. Guidelines for Design and Development of Network Functions

Ideally, as outlined in Section II, the VNF ecosystem should be designed based on a detailed performance requirement specification for the intended deployment scenario. However, due to internal policies or previous design choices, the essential execution environment is often already given when the performance becomes constraining. In the following, we concentrate on this deployment optimization scenario when the task is to improve VNF performance for a given environment. We highlight best practices for the VNF adoption, derive corresponding suggested implementation *patterns*, and summarize caveats in *antipatterns*, which should be avoided.

1) Know relevant performance metrics: To bootstrap the performance optimization process, it is necessary to be aware of the network functions, the current workload, the resource utilization of the whole ecosystem, as well as the underlying network topology.

✓ *Pattern*: Deployment of dedicated monitoring and verification tools providing up-to-date reports about the actual performance and resource descriptors in the system.

✗ *Antipattern*: Inaccurate view due to missing monitoring information and insufficient insights may result in resource over-provisioning, e.g., by vertical scaling.

2) Optimize for the most constraining resource: Apply the “Pareto-principle” of software optimization to identify the 10% of the code affecting the most critical resource (e.g., most of the running time is spent, most of the memory is used) and apply the previously outlined software and hardware acceleration techniques. Table I is a good starting point to identify the bottleneck resources for the specific VNF type.

✓ *Pattern*: Repeat the following performance optimization loop: (1) Identify per-VNF critical resources; (2) obtain deployment-specific insights using monitoring tools; (3) optimize for the most constraining resource by choosing appropriate software or hardware acceleration; (4) test and evaluate the results.

✗ *Antipattern*: Rewriting a VNF for using a specific acceleration technique will improve *one specific bottleneck resource*. For instance, [102] shows that SGW acceleration with DPDK improves raw packet-per-second performance while user space table lookup remains a bottleneck, resulting in poor performance as soon as the table is sufficiently populated.

3) Exploit parallelism: Network-related workloads typically lend themselves readily to a parallel pipelined execution model, by providing ample opportunities to separate a workload to multiple threads at the packet level, at the protocol level, or at the network-function level. Network-related workloads can be balanced among multiple threads at packet, protocol, or network function level. Thus, it is possible to utilize the parallel execution capabilities of modern hardware devices to execute atomic building blocks in parallel.

✓ *Pattern*: Build and optimize network functions to maximize concurrency, by minimizing contention on shared resources and point-to-point synchronization (coherency) between execution threads.

✗ *Antipattern*: Writing code that is fundamentally serial in nature causes significant runtime overhead due to synchron-

ization between execution threads. Similarly, resource-hungry synchronization primitives or frequent memory accesses across NUMA nodes constitute performance bottlenecks.

4) Use lightweight virtualization & isolation: The choice of the specific virtualization technique is a careful balancing act between flexibility, performance, overhead, and degree of isolation with respect to CPU, network, and storage resources.

✓ *Pattern*: Use of the lightest possible isolation technique that is still sufficient to provide the required isolation barrier. This might also include higher layer isolation techniques for safe zero-cost isolation. Examples are compile-time type checking and a safe runtime enabled purely in software [103].

✗ *Antipattern*: Using lightweight virtualization across trust boundaries, e.g., deploying sensitive applications of different tenants into containers with limited isolation capabilities.

5) Cautious use of hardware offloading: Hardware offloading is very appealing since it allows to perform certain CPU-intensive operations in hardware, thereby relieving the CPU. Possible drawbacks, however, are a lack of isolation as well as additional I/O costs. Furthermore, the network topology should be taken into account since hardware offloading capabilities might be limited to a subset of network nodes.

✓ *Pattern*: Careful measurement of critical and reusable software building blocks that can be effectively realized in hardware and a step-by-step process to apply offloading gradually to CPU-intensive code.

✗ *Antipattern*: Using hardware offloading in the middle of a service chain may involve an unnecessary I/O roundtrip to hardware, possibly slowing execution down instead of accelerating it.

B. Guidelines for Software Acceleration

Based on our expertise in the software networking domain, we provide guidelines for the development of VNFs. From the point of view of a developer, two points should be kept in mind. First, a trade-off may arise between the acceleration benefit and the complexity of the implementation of a specific acceleration technique. Second, some of the acceleration techniques work better in conjunction with some classes of virtual network functions. We provide a brief list of good practices and develop each point in the following.

1) Utilize hardware-supported functions when available: Hardware-supported functions can have a significant performance impact and can be easily introduced into existing software due to widespread driver integration. The two most important examples from this category include hardware counters and RSS hash functions. While using the former can significantly reduce the amount of state and memory accesses, the latter lowers the number of CPU cycles.

2) Build virtual network functions on top of general-purpose accelerated frameworks: Nowadays, high-speed frameworks exploit several state-of-the-art acceleration techniques and also provide mechanisms for interconnecting several VNFs at high speed. Hence, developers can focus on their particular application while leveraging the infrastructure and flexibility that is offered by such frameworks. The choice of the specific framework in the wide ecosystem may be dictated

by the environment (e.g., pre-existing expertise) or by specific framework capabilities (e.g., availability of specific functions), or finally by performance considerations (i.e., which is harder as of now, since a comprehensive comparison of the available frameworks is still an open research question).

3) Implement very specific tasks using low-level building blocks for CPU-intensive tasks, and leverage advanced coding styles: Finally, when computation-intensive VNFs have to be deployed on bare metal or on top of an existing high-speed framework, there may be cases in which the I/O is dominated by the computation, or situations where the bottleneck of I/O is negligible. For an overloaded system, i.e., I/O and CPU bottlenecks in cascade, polling and I/O batching can accelerate the packet processing. More generally, memory-related acceleration, compute batching, and high-level programming techniques are beneficial for the VNF (paying attention to the zero-copy approach, that can be useful when there is no further payload modification). Collectively, the use of these techniques allow to optimize CPU processing of VNFs in software.

C. Guidelines for Hardware Acceleration

Due to the required amount of expert knowledge in different domains such as programming languages, hardware architectures, and verification mechanisms, the design of hardware accelerated functions requires significantly more effort than the corresponding process for their software counterpart. Therefore, the choice of offloading a network function, or part of it, to a SmartNIC is a careful decision that depends on several factors that we summarize in the following.

1) Performance gain: Offloading a primitive into the NIC is recommended when the processing power, e.g., in terms of CPU cycles, saved by doing so is significant.

2) Primitive adoption: The widespread use of a primitive is an important factor when deciding whether it is worth to implement it in the NIC.

3) Complexity: Due to the high cost of non-recurring engineering of hardware development, the decision about the offloading heavily depends on the complexity of the primitive. Furthermore, the expected frequency of updates to the functionality of network functions should be taken into account due to the fact that each update induces a development overhead. Finally, the presence of temporal dynamics that require frequent migration of VNFs should be carefully considered since the migration process tends to be more complex with hardware-accelerated components in contrast to purely software-based ones.

4) Relation of the primitive to other processing functions: Offloading to the NIC is only useful if the subsequent operations are also offloaded to the NIC. Otherwise, costly back and forth movements between the NIC and the host are required.

Finally, it is worth noting that the use of network-focused programming abstractions for the design of hardware accelerated functions can greatly simplify the development cycle. In our opinion, this is a important and still immature research field that could provide a significant advance in network development in the future. This is particularly true when dealing with issues associated with running multiple applications using the same accelerator and multi-tenancy.

SmartNICs are one of the most discussed and evolving technologies for accelerating network functions. As mentioned in Section IV-B there are two main SmartNIC types. Multicore SoC-based and FPGA-based.

Designing efficient network functions using Multicore NICs requires to deal with some typical issues of parallel programming. The single core of the SoC can be programmed using standard programming languages (mainly C/C++ is used) and does not present specific challenging issues. On the other hand, the interaction among the cores, the access to shared resources and the efficient use of the AEs as well as the distribution of the packet among the cores are hard to manage and often requires a deep knowledge of the specific hardware architecture of the SoC. This makes the implementation very target specific, preventing the porting to a different SoC-based SmartNIC, and slows down the development of efficient network functions.

When FPGA-based SmartNICs are programmed using low level HDL, they demand extremely long development and verification time. Thus, even if in terms of absolute performance they could provide better throughput than SoC-based SmartNICs, the design effort could make it more convenient to use more programmable friendly SoC-based SmartNICs. If the programming abstractions described in Section IV-C will evolve to mature and easy to use programming paradigm, the FPGA-based SmartNIC will emerge as the right element to provide programmable hardware accelerators for NFs.

From the performance point of view, SmartNICs, in particular FPGA-based ones, provide benefits for both latency and throughput. For instance, [90] reports a 3-5x improvement in terms of average and worst case latency as well as a sixfold increase in throughput when employing an FPGA-based SmartNIC. While these gains have to be interpreted as of anecdotal value, since they relate about a performance comparison of a specific design, they nevertheless are useful to report, to perceive the expected gains in “order sense” than the offloading of complex functions to a SmartNIC can bring. In terms of economic advantage in using SmartNICs, one should consider that such devices can free valuable CPU cores for other tasks. For instance, Microsoft reports that in a cloud datacenter each core has an equivalent value of about 800\$ per year [90].

Finally, there are cases in which using a FPGA-based SmartNIC provides much larger benefits. This is the case for instance, of applications that require very low-latency and/or predictable performance, such as high-performance trading. Another area where SmartNICs can be beneficial is deployments that are somehow space constrained, e.g., in an edge datacenter deployed in a big city, where real estate costs are particularly expensive. In these cases, either the other approaches are completely ruled out by unmet requirements (e.g., latency below 1us), or the application-level and the overall infrastructure-level advantages offset the extra cost paid in buying a powerful SmartNIC.

D. Potential of Acceleration Techniques for the NGFW

In addition to the above guidelines regarding software-and hardware-based acceleration techniques, we present an

overview of insights that are relevant to the NGFW VNF. To this end, we first recall the building blocks that are associated with this VNF in Table I and then illustrate how we can improve their performance. We note, however, that since the main focus of this manuscript is not the NGFW, we do not provide an exhaustive set of guidelines and options but exemplary demonstrate how three building blocks might be improved.

On the one hand, the NGFW comes with cryptography support and therefore needs to perform compute-intensive decryption and encryption tasks on packet payloads. As outlined in Section IV-A2, significant amounts of CPU time can be saved by offloading these tasks to a hardware entity. Depending on the capabilities in terms of supported encryption standards, this entity might be a commodity NIC, a specialized NPU, or an FPGA component inside the NIC.

On the other hand, the NGFW has NAPT and VPN capabilities which require rewriting packet headers. In this context, switching from plain virtualization to software acceleration via frameworks like DPDK has been shown to significantly improve performance [104]. In the referenced work, we compare the performance of an LTE Serving Gateway VNF that is implemented as a user-space application with a DPDK-accelerated version of the same VNF. We demonstrate a throughput improvement of almost factor 8 when using the latter. These results can be translated to the NGFW since both functions share the corresponding building block.

Furthermore, the NGFW employs DPI techniques in its decision making process which require tracking flow and session states as well as processing header and payload information. Whereas, as previously introduced we deem a thorough account of DPI acceleration via FPGA or GPUs to be out of the scope of this work, it is nevertheless instructive to consider how the software guidelines expressed above can boost the performance of a “vanilla” DPI application in the NGFW. Indeed, even though no packet modification is performed during these tasks, significant improvements in terms of maximum throughput are possible when using SR-IOV-based virtualization in conjunction with an acceleration framework such as DPDK [102]. When compared to an implementation based on the default Linux kernel stack and a libpcap-based DPI application, an up to tenfold throughput increase is observed. While, again, these gains have to be interpreted as of anecdotal value, they relate the expected gains in “order sense” than adopting state-of-the art software techniques can bring.

VI. CONCLUSION

The rising amount of networked devices and services, the heterogeneity of their demands and communication patters, as well as their temporal dynamics pose stringent requirements that exceed the capabilities of today’s networking architectures. These phenomena call for a new type of scalable and adaptive systems. Network softwarization, which comprises networking paradigms such as Software Defined Networking (SDN) and Network Functions Virtualization (NFV), is seen as a possible solution to fulfill these requirements and to

provide a scalable and adaptive networking infrastructure. As NFV provides the necessary degree of flexibility by moving today’s embedded network functions to software running on common server hardware, recent efforts try to improve the performance of these software-based solutions.

In this work, we conduct a survey of the evolving research field of performance acceleration techniques for network functions virtualization. First, we categorize a subset of representative network functions with respect to common low-level building blocks. In conjunction with information on the resource usage of individual building blocks, we identify the corresponding performance bottlenecks.

After that, we review and summarize software-based and hardware-based acceleration techniques in a holistic manner. We first highlight pure software acceleration techniques allowing to speed up the the performance of a network function itself. After that, we cover the software stack as a whole and discussed the interplay between software and driver-based hardware access. In the case of hardware acceleration techniques, we cover offloading strategies based on standard NICs, programmable SmartNICs, as well as domain-specific programming abstractions. These mechanisms represent trade-offs w.r.t. the implementation overhead, performance gains, and costs.

Based on the survey, we finally derive guidelines which allow the reader to understand which acceleration techniques might help to improve the performance of specific network functions. Our summary includes generic guidelines that represent best practices when optimizing the performance of a given deployment scenario, as well as specific guidelines on how to apply software and hardware acceleration techniques. Our paper thus gives guidance to potential NFV adopters on how to improve the performance of their current system, clearly highlighting the research direction that are still open and require a community wide attention.

VII. ACKNOWLEDGEMENT

This work benefited from support of NewNet@Paris, Cisco’s Chair “NETWORKS FOR THE FUTURE” at Telecom ParisTech (<http://newnet.telecom-paristech.fr>). This work has been performed in the framework of the CELTIC EUREKA project SENDATE-PLANETS (Project ID C2015/3-1), and it is partly funded by the German BMBF (Project ID 16KIS0474), the BMBF funded project Berlin Big Data Center BBDC (01IS14013A), and by the European Commission in the frame of the Horizon 2020 project 5G-PICTURE (grant #762057). Gábor Rétvári was supported by the MTA-BME Network Softwarization Research Group, at the Budapest University of Technology and Economics. Any opinion, findings or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of partners of the Chair and other funding entities.

REFERENCES

- [1] ETSI GR NFV 001 - Use Cases, ETSI Group Report, Rev. 1.2.1, 2017.
- [2] B. Yi, X. Wang, K. Li, M. Huang *et al.*, "A comprehensive survey of network function virtualization," *Computer Networks*, 2018.
- [3] B. Carpenter and S. Brim, "Middleboxes: Taxonomy and issues," RFC 3234, 2002.
- [4] M. Hoffmann, M. Jarschel, R. Pries, P. Schneider, A. Jukan, W. Bziuk, S. Gebert, T. Zinner, and P. Tran-Gia, "SDN and NFV as Enabler for the Distributed Network Cloud," *Mobile Networks and Applications*, 2017.
- [5] ETSI GS NFV-PER 001 - NFV Performance & Portability Best Practices, ETSI Group Specification, Rev. 1.1.2, 2014.
- [6] The Bro Network Security Monitor. [Online]. Available: <https://www.bro.org/>
- [7] Snort IDS. [Online]. Available: <https://www.snort.org/>
- [8] M. Trevisan, A. Finamore, M. Mellia, M. Munafò, and D. Rossi, "Traffic Analysis with Off-the-Shelf Hardware: Challenges and Lessons Learned," *IEEE Communication Magazine*, 2017.
- [9] V. Addanki, L. Linguglossa, J. W. Roberts, and D. Rossi, "Controlling software router resource sharing by fair packet dropping," in *IFIP Networking*, 2018.
- [10] L. Deri, M. Martinelli, T. Buijlow, and A. Cardigliano, "nDPI: Open-source high-speed deep packet inspection," in *International Wireless Communications and Mobile Computing Conference (IWCMC)*, 2014.
- [11] ETSI GS NFV-IFA 001 - Report on Acceleration Technologies & Use Cases, ETSI Group Specification, Rev. 1.1.1, 2015.
- [12] J. Sahoo, S. Mohapatra, and R. Lath, "Virtualization: A survey on concepts, taxonomy and associated security issues," in *International Conference on Computer and Network Technology (ICCNT)*, 2010.
- [13] 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 *ACM SIGOPS operating systems review*, 2003.
- [14] VMWare ESXi. [Online]. Available: <https://www.vmware.com/products/esxi-and-esx.html>
- [15] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Ligouri, "KVM: the linux virtual machine monitor," in *Proceedings of the Linux symposium*, 2007.
- [16] J. Hwang, S. Zeng, F. Wu, and T. Wood, "A component-based performance comparison of four hypervisors," in *IM*, 2013.
- [17] G. Lettieri, V. Maffione, and L. Rizzo, "A Survey of Fast Packet I/O Technologies for Network Function Virtualization," in *International Conference on High Performance Computing*, 2017.
- [18] R. Russell, "virtio: towards a de-facto standard for virtual I/O devices," *ACM SIGOPS Operating Systems Review*, 2008.
- [19] V. Maffione, L. Rizzo, and G. Lettieri, "Flexible virtual machine networking using netmap passthrough," in *IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, 2016.
- [20] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, 2014.
- [21] P.-H. Kamp and R. N. Watson, "Jails: Confining the omnipotent root," in *International SANE Conference*, 2000.
- [22] A. Madhavapeddy and D. J. Scott, "Unikernels: the rise of the virtual library operating system," *Communications of the ACM*, 2014.
- [23] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2017.
- [24] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski *et al.*, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*, 2017.
- [25] N. M. K. Chowdhury and R. Boutaba, "A survey of network virtualization," *Computer Networks*, 2010.
- [26] A. Blenk, A. Basta, M. Reisslein, and W. Kellerer, "Survey on network virtualization hypervisors for software defined networking," *IEEE Communications Surveys & Tutorials*, 2016.
- [27] ETSI GS NFV 002 - Architectural Framework, ETSI Group Specification, Rev. 1.2.1, 2014.
- [28] N. F. S. de Sousa, D. A. L. Perez, R. V. Rosa, M. A. Santos, and C. E. Rothenberg, "Network service orchestration: A survey," *arXiv preprint arXiv:1803.06596*, 2018.
- [29] H. Guan, Y. Dong, R. Ma, D. Xu, Y. Zhang, and J. Li, "Performance enhancement for network I/O virtualization with efficient interrupt coalescing and virtual receive-side scaling," *IEEE Transactions on Parallel and Distributed Systems*, 2013.
- [30] L. Rizzo, "Device polling support for FreeBSD," in *BSDConEurope Conference*, 2001.
- [31] The Linux Foundation, "Wiki: napi." [Online]. Available: <https://wiki.linuxfoundation.org/networking/napi>
- [32] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: a GPU-accelerated software router," in *SIGCOMM*, 2010.
- [33] J. Kim, S. Huh, K. Jang, K. Park, and S. Moon, "The power of batching in the click modular router," in *Asia-Pacific Workshop on Systems*, 2012.
- [34] L. Linguglossa *et al.*, "High-speed Software Data Plane via Vectorized Packet Processing (Extended Version)," Telecom ParisTech, Tech. Rep., 2017.
- [35] "Data plane development kit." [Online]. Available: <http://dpdk.org>
- [36] L. Rizzo, "netmap: a novel framework for fast packet I/O," in *USENIX ATC*, 2012.
- [37] Deri, L. [Online]. Available: [https://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zczero-copy/](https://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/)
- [38] G. Rétvári, J. Tapolcai, A. Kőrösi, A. Majdán, and Z. Heszberger, "Compressing IP forwarding tables: Towards entropy bounds and beyond," in *ACM SIGCOMM*, 2013.
- [39] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *ANCS*, 2015.
- [40] L. Molnár, G. Pongrácz, G. Enyedi, Z. L. Kis, L. Csikor, F. Juhász, A. Kőrösi, and G. Rétvári, "Dataplane specialization for high performance OpenFlow software switching," in *ACM SIGCOMM*, 2016.
- [41] A. Beaupré, "New approaches to network fast paths," 2017. [Online]. Available: <https://lwn.net/Articles/719850/>
- [42] The Linux Foundation, "eXpress Data Path (XDP)." [Online]. Available: <https://www.iovisor.org/technology/xdp>
- [43] J. Corbet, "Zero-copy networking," *Linux Weekly News*, 2017.
- [44] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen, "Raising the bar for using gpus in software packet processing," in *NSDI*, 2015.
- [45] R. Mansilha, L. Saino, M. Barcellos, M. Gallo, E. Leonardi, D. Perino, and D. Rossi, "Hierarchical Content Stores in High-speed ICN Routers: Emulation and Prototype Implementation," in *ICN*, 2015.
- [46] G. Xylomenos, C. N. Verteridis, V. A. Siris, N. Fotiou, C. Tsilopoulos, X. Vasilakos, K. V. Katsaros, and G. C. Polyzos, "A survey of information-centric networking research," *IEEE Communication Surveys & Tutorials*, 2014.
- [47] I. Marinos, R. N. M. Watson, and M. Handley, "Network stack specialization for performance," in *SIGCOMM*, 2014.
- [48] T. Zhang, L. Linguglossa, M. Gallo, P. Giaccone, and D. Rossi, "Flowmon-dpdk: Parsimonious per-flow software monitoring at line rate," in *IFIP Traffic Monitoring and Analysis*, 2018.
- [49] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems," in *NSDI*, 2014.
- [50] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of Open vSwitch," in *USENIX NSDI*, 2015.
- [51] E. Kohler, R. Morris, B. Chen, J. Jannotti, and F. Kaashoek, "The Click Modular Router," *Operating Systems Review*, 1999.
- [52] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "SoftNIC: a software NIC to augment hardware," EECS Department, University of California, Berkeley, Tech. Rep., 2015.
- [53] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *USENIX Conference on Networked Systems Design and Implementation*, 2014.
- [54] M. Gallo and R. Laufer, "Clicknf: a modular stack for custom network functions," in *USENIX Annual Technical Conference (ATC)*, 2018.
- [55] M. A. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park, "mos: A reusable networking stack for flow monitoring middleboxes," in *NSDI*, 2017.
- [56] A. Abdelrazik, G. Bunce, K. Cacciatore, K. Hui, S. Mahankali, and F. Van Rooyen, "Adding speed and agility to virtualized infrastructure with openstack," 2017.
- [57] J. Hwang, K. K. Ramakrishnan, and T. Wood, "Netvm: high performance and flexible networking using virtualization on commodity platforms," *IEEE Transactions on Network and Service Management*, 2015.
- [58] A. Belay, G. Prekas, M. Primorac, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "The ix operating system: Combining low latency, high throughput, and efficiency in a protected dataplane," *ACM Transactions on Computer Systems (TOCS)*, 2017.
- [59] S. Peter, T. Anderson, and T. Roscoe, "Arrakis: The Operating System as Control Plane," *ACM Transactions on Computer Systems*, 2013.

- [60] N. Pitaev, M. Falkner, A. Leivadeas, and I. Lambadaris, "Characterizing the performance of concurrent virtualized network functions with OVS-DPDK, FD.IO VPP and SR-IOV," in *ACM/SPEC International Conference on Performance Engineering*, 2018.
- [61] N. Zilberman, P. M. Watts, C. Rotsos, and A. W. Moore, "Reconfigurable network systems and software-defined networking," *Proceedings of the IEEE*, 2015.
- [62] Intel, "Intel 10 Gigabit AT Server Adapter." [Online]. Available: <https://www.intel.com/content/dam/support/us/en/documents/network/adapter/10gbe/atserver/sb/318349.pdf>
- [63] Mellanox, "ConnectX-6 EN IC 200Gb/s Ethernet Adapter IC." [Online]. Available: http://www.mellanox.com/related-docs/prod_silicon/PB_ConnectX-6_EN_IC.pdf
- [64] ETSI GS NFV-IFA 002 - *Network Functions Virtualisation (NFV); Acceleration Technologies; VNF Interfaces Specification*, ETSI Group Specification, Rev. 2.1.1, 2017.
- [65] "openNFP," 2016. [Online]. Available: <http://open-nfp.org/>
- [66] Netronome, "AgilioTM CX 2x40GbE intelligent server adapter" [Online]. Available: https://www.netronome.com/media/redactor_files/PB_Agilio_CX_2x40GbE.pdf
- [67] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "NetFPGA SUME: Toward 100 Gbps as Research Commodity," *IEEE Micro*, 2014.
- [68] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.
- [69] A. Kaufmann, S. Peter, T. Anderson, and A. Krishnamurthy, "FlexNIC: rethinking network DMA," in *USENIX HotOS*, 2015.
- [70] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey, "Architecting to achieve a billion requests per second throughput on a single key-value store server platform," in *Annual International Symposium on Computer Architecture*, 2015.
- [71] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman, "Be fast, cheap and in control with switchkv," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.
- [72] H. Ballani, P. Costa, C. Gkantsidis, M. P. Grosvenor, T. Karagiannis, L. Koromilas, and G. O'Shea, "Enabling end-host network functions," *SIGCOMM Computer Communications Review*, 2015.
- [73] D. Firestone, "VFP: A virtual switch platform for host SDN in the public cloud," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.
- [74] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon, "P4fpga: A rapid prototyping framework for p4," in *Symposium on SDN Research*, 2017.
- [75] G. Brebner and W. Jiang, "High-speed packet processing using reconfigurable computing," *IEEE Micro*, 2014.
- [76] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis, "Regular expression matching on graphics hardware for intrusion detection," in *International Workshop on Recent Advances in Intrusion Detection*, 2009.
- [77] S. Pontarelli, G. Bianchi, and S. Teofili, "Traffic-aware design of a high-speed FPGA network intrusion detection system," *IEEE Transactions on Computers*, 2013.
- [78] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep packet inspection using parallel bloom filters," in *Symposium on high performance interconnects*, 2003.
- [79] F. Yu, Z. Chen, Y. Diao, T. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *ACM/IEEE Symposium on Architecture for Networking and Communications systems*, 2006.
- [80] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using fpgas," in *Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.
- [81] R. Antonello, S. Fernandes, C. Kamienski, D. Sadok, J. Kelner, I. GóDor, G. Szabó, and T. Westholm, "Deep packet inspection tools and techniques in commodity platforms: Challenges and trends," *Journal of Network and Computer Applications*, 2012.
- [82] Intel, "Intel Pro/100S Network Adapters. IPsec Offload Performance and Comparison." [Online]. Available: <https://www.intel.com/content/dam/doc/performance-brief/testing-labs-report-intel-pro-100-s-network-adapters-brief.pdf>
- [83] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy, "High-resolution measurement of data center microbursts," in *Proceedings of the 2017 Internet Measurement Conference*. ACM, 2017, pp. 78–85.
- [84] S. Pontarelli, G. Bianchi, and M. Welzl, "A programmable hardware calendar for high resolution pacing," in *High Performance Switching and Routing (HPSR), 2018 IEEE 19th International Conference on*. IEEE, 2018.
- [85] M. G. Adrian Caulfield, Paolo Costa, "Beyond smartnics: Towards a fully programmable cloud," in *IEEE International Conference on High Performance Switching and Routing*, 2018.
- [86] "Cavium LiquidIO II Network Appliance Smart NICs." [Online]. Available: http://www.cavium.com/LiquidIO-II_Network_Appliance_Adapters.html
- [87] "BlueField Multicore System on Chip." [Online]. Available: http://www.mellanox.com/related-docs/npu-multicore-processors/PB_Bluefield_SoC.pdf
- [88] "Open vSwitch Offload and Acceleration with Agilio CX SmartNICs." [Online]. Available: https://www.netronome.com/media/redactor_files/WP_OVS_Benchmarking.pdf
- [89] "BlueField Multicore System on Chip." [Online]. Available: http://www.mellanox.com/related-docs/prod_multi_core/PB_TILE-Gx72.pdf
- [90] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung *et al.*, "Azure Accelerated Networking: SmartNICs in the Public Cloud," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.
- [91] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *ACM SIGCOMM CCR*, 2008.
- [92] Intel, "Intel FlowDirector." [Online]. Available: <https://software.intel.com/en-us/articles/setting-up-intel-ethernet-flow-director>
- [93] P. Bosschart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *ACM SIGCOMM*, 2013.
- [94] R. Duncan and P. Jungck, "packetc language for high performance packet processing," in *IEEE International Conference on High Performance Computing and Communications (HPCC)*, 2009.
- [95] H. Song, "Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane," in *ACM SIGCOMM workshop on Hot topics in software defined networking*, 2013.
- [96] "Linux Socket Filtering aka Berkeley Packet Filter (BPF)." [Online]. Available: <https://www.kernel.org/doc/Documentation/networking/filter.txt>
- [97] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking, "Packet transactions: High-level programming for line-rate switches," in *ACM SIGCOMM*, 2016.
- [98] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: Programming platform-independent stateful openflow applications inside the switch," *ACM SIGCOMM CCR*, 2014.
- [99] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan, "Flow-level state transition as a new switch primitive for sdn," in *Workshop on Hot Topics in Software Defined Networking*, 2014.
- [100] G. Bianchi, M. Bonola, S. Pontarelli, D. Sanvito, A. Capone, and C. Cascone, "Open packet processor: a programmable architecture for wire speed platform-independent stateful in-network processing," *arXiv preprint*, 2016. [Online]. Available: <http://arxiv.org/abs/1605.01977>
- [101] A. Caulfield, P. Costa, and M. Ghobadi, "Beyond SmartNICs: Towards a Fully Programmable Cloud," in *IEEE HPSR*, 2018.
- [102] M.-A. Kourtis, G. Xilouris, V. Riccobene, M. J. McGrath, G. Petralia, H. Koumaras, G. Gardikis, and F. Liberal, "Enhancing VNF performance by exploiting SR-IOV and DPDK packet processing acceleration," in *IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, 2015.
- [103] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the V out of NFV," in *USENIX OSDI*, 2016.
- [104] S. Lange, A. Nguyen-Ngoc, S. Gebert, T. Zinner, M. Jarschel, A. Koepsel, M. Sune, D. Raumer, S. Gallenmüller, G. Carle, and P. Tran-Gia, "Performance Benchmarking of a Software-Based LTE SGW," in *International Workshop on Management of SDN and NFV Systems (ManSDN/NFV)*, 2015.



Leonardo Linguaglossa Leonardo Linguaglossa is a post-doctoral researcher at Télécom ParisTech (France). He received his master degree in telecommunication engineering at University of Catania (Italy) in 2012. He pursued a Ph.D. in Computer Networks in 2016 through a joint doctoral program with Alcatel-Lucent Bell Labs (nowadays Nokia), INRIA and University Paris 7. Leonardo's research interests focus on architecture, design and prototyping of systems for high-speed software packet processing, future Internet architecture and SDN.



Thomas Zinner Thomas Zinner is visiting professor and head of the research group "Internet Network Architectures" at the Technical University of Berlin, Germany, since 2018. He received his PhD in 2012 and his professorial thesis (habilitation) in 2017 from the University of Würzburg. He has published more than 80 research papers in major conferences and journals, receiving six best paper and best student paper awards. His research focuses on QoE management, network softwarization and programmable data planes, and performance evaluation.



Stanislav Lange Stanislav Lange studied computer science at the University of Würzburg, Germany, where he received his MSc degree in 2014. Currently, he is a researcher in the "Next Generation Networks" research group at the Chair of Communication Networks in Würzburg and is pursuing his PhD. His research is focused on software defined networking, performance analysis, system modeling, as well as multiobjective optimization.



Roberto Bifulco Roberto Bifulco is a senior researcher in the Systems and Machine Learning group at the NEC Laboratories Europe. His main research interests are the design of high-performance networked systems and the exploration of novel solutions at the crossroad of systems and machine learning. Before joining NEC in 2012, he worked as consultant for small technological enterprises and start-ups in the field of server virtualization and cloud computing. Roberto holds a Ph.D. from University of Napoli "Federico II", Italy.



Salvatore Pontarelli Salvatore Pontarelli received a master degree in electronic engineering at University of Bologna and the PhD degree in Microelectronics and Telecommunications from the University of Rome Tor Vergata. Currently, he works as Senior Researcher at CNIT (Italian National Inter-University Consortium for Telecommunications), in the research unit of University of Rome Tor Vergata. His research interests include hash based structures for networking applications, use of FPGA for high speed network monitoring and hardware design of software defined network devices.



Michael Jarschel Michael Jarschel is working as a research engineer in the area of Network Softwarization and Connected Driving at Nokia Bell Labs in Munich, Germany. He finished his Ph.D. thesis, titled "An Assessment of Applications and Performance Analysis of Software Defined Networking", at the University of Würzburg in 2014. His main research interests are in the applicability of SDN and NFV concepts to next generation networks as well as distributed telco cloud technologies and their use cases.



Gábor Rétvári Gábor Rétvári received the M.Sc. and Ph.D. degrees in electrical engineering from the Budapest University of Technology and Economics in 1999 and 2007. He is now a Senior Research Fellow at the Department of Telecommunications and Media Informatics. His research interests include all aspects of network routing and switching, the programmable data plane, and the networking applications of computational geometry and information theory. He maintains several open source scientific tools written in Perl, C, and Haskell.



Giuseppe Bianchi Giuseppe Bianchi is Full Professor of Networking at the Univ. of Roma Tor Vergata. His research activity includes programmable network systems, wireless networks, privacy and security, traffic control. He has been general or technical co-chair for several major IEEE and ACM conferences and workshops, and has held general or technical coordination roles in six European projects so far. He has been editor for IEEE/ACM Trans. on Networking, IEEE Trans. on Wireless Commun., IEEE Trans. on Network and Service Management, and Elsevier Computer Communications.



Dario Rossi Dario Rossi received his MSc and PhD degrees from Politecnico di Torino in 2001 and 2005 respectively, and his HDR degree from Université Pierre et Marie Curie (UPMC) in 2010. He is currently a Professor at Telecom ParisTech and Ecole Polytechnique and is the holder of Cisco's Chair NewNet@Paris. He has coauthored 9 patents and over 150 papers in leading conferences and journals, that received 8 best paper awards, a Google Faculty Research Award (2015) and an IRTF Applied Network Research Prize (2016). He is a Senior