Software Acceleration Techniques for High-speed Programmable Networks

Leonardo Linguaglossa

Software acceleration techniques for high-speed programmable networks

Leonardo Linguaglossa

December 2020

This document is a preprint version of chapter 11 of the CNIT Technical Report 06 titled *"Network Programmability: a (r)evolutionary approach"*. The full Report can be purchased at the following URL:

https://www.texmat.it/06-network-programmability-a-r-evolutionary-approach.html

When citing this work, **please cite the original book**:

Linguaglossa L. (2020). Software acceleration techniques for high-speed programmable networks. In Bianchi G., Cerroni W., Palazzo S., "Network Slicing. Network Programmability: a (r)evolutionary approach" (Chapter 11, pp. 203-223). ISBN 9788894982428.

Software Acceleration Techniques for High-speed Programmable Networks

Leonardo Linguaglossa[1]

[1]Telecom Paris - France

Abstract: *Network programmability has provided an effective approach to enable innovation in network systems. By replacing static, expensive middleboxes with equivalent pieces of software implementing the same functionality, operators can significantly reduce their CAPEX/OPEX expenditures, and engineers can rapidly design, test and deploy novel architectures and services, thus reducing the time-to-market for network applications. However, the flexibility provided by software solutions comes at a cost: purpose-specific hardware has the clear advantage of optimized performance (i.e., throughput, latency) with respect to pure software-based solutions. The introduction of software acceleration techniques represented an essential step towards the increasing popularity of the SDN/NFV paradigm shift, by reducing the performance gap between hardware-based and software-based systems. Thanks to such techniques, modern software-networking solutions can operate at multi-10Gbps rate (up to hundreds of Gbps) on commodity servers equipped with regular COTS components. In this chapter, we cover the aspects related to software acceleration techniques in a bottom-up fashion: we first provide an overview of high-speed software networking on COTS architectures, and we then explore the evolution of softwarized networking by focusing on performance acceleration and the design space for high-speed programmable networks.*

1 Introduction

In the last decades, the networking industry had experienced a major architectural shift towards the softwarization of network devices [1]. Besides the growing popularity of programmable hardware such as OpenFlow switches [2, 3] or P4 switches [4, 5], commonly adopted Network Interface Cards (NICs) are being replaced with more programmable counterparts known as SmartNICs [6]. In line with this trend, network functions are being executed in pure software on top of commodity servers, rather than using static expensive middleboxes [7].

Together with the growing expansion of cloud appliances [8], current trends show a tremendous amount of network functionalities being executed via software instead of hardware. This context has provided the strong foundation for the evolution of novel network paradigms such as Software-defined Networking (SDN), or Network Function Virtualization (NFV). Some efforts to bring software-programmability in the network environment date back to the end of the twentieth century, as for the Click modular router [9], one of the first frameworks that allowed users to map network functions into pieces of software, thus originating the shift from hardware-specific components to more

flexible alternatives run on COTS equipment. The main idea of Click is to provide a set of C++ libraries used to create standard or custom network functions. A simple programming language can be used to link such functions, connect them to external NICs and specify the packet processing workflow in a graph-like fashion, realizing a software modular router instance.

As we shall see in this chapter, several concepts introduced by Kohler's Click are still relevant in modern solutions for high-speed packet processing: from the focus on COTS general-purpose hardware to deploy software routers and elude the network ossification problem, to the idea of a "forwarding graph" to specify the desired packet processing. However, software solutions had to face a natural trade-off: with high flexibility comes a significant performance gap with respect to hardware solutions. Although classical network applications relied on the support of the operating system for both the management and the data plane (e.g., retrieving packets from the network card, allocating memory to store packets, scheduling the functions to be executed on a CPU, perform some processing in the TCP/IP stack, ...), nowadays NICs capabilities can reach 100 Gbps rate and beyond [10], a 100x improvement w.r.t. to 20 years ago; this operational point is too high for general-purpose OS kernels to sustain [11]. These limitations have not stopped the desire for a flexible approach based on pure software for implementing network functionalities: on the contrary, they boosted the research effort in the direction of reducing the gap between hardware and software systems for packet processing [12]. In this chapter, we focus on the techniques utilized to accelerate the packet processing on COTS systems. We first provide a general overview on the components of a COTS architecture (Sec. 2). Then we present the most important acceleration techniques used by state-of-the-art high-speed packet processing engines (Sec. 3). Finally, we conclude the chapter with some considerations about the design space of accelerated frameworks in Sec. 4.

2 High-speed software networking on COTS servers

The need for flexibility and programmability in network environments has fueled the ongoing process of "network softwarization" and inspired the development of novel paradigms (among which the aforementioned SDN and NFV) that share some common concepts: decoupling the forwarding plane from the control plane, the separation of network functions from the underlying hardware, virtualization of network resources [13]. Most of such solutions are deployed on commercial off-the-shelf (COTS) servers, which are commodity platforms equipped with general-purpose multi-core processors belonging to the x86 [14, 15] or ARM [16] families, often connected to the memory through a Non-Uniform Memory-Access (NUMA) architecture [17]. These servers are managed by an operating system (typically a Linux flavor) and can access one (or more) NIC(s) through a specific network device driver. A schematic representation of this architecture is shown in Figure 1. In a top-down view, several applications of one or more users can coexist at user-space. Each application can be a stand-alone process or a virtual machine managed by a hypervisor. Network-related apps can access the NICs either (i) via the mediation of the OS kernel (which manages the processing units and the memory as well); or (ii) through some user-space libraries which bypass the operating system drivers; or (iii) by implementing from scratch a full kernel-bypass application. Most of the past work in this domain focused on edge systems deployed within COTS equipment [18], but this

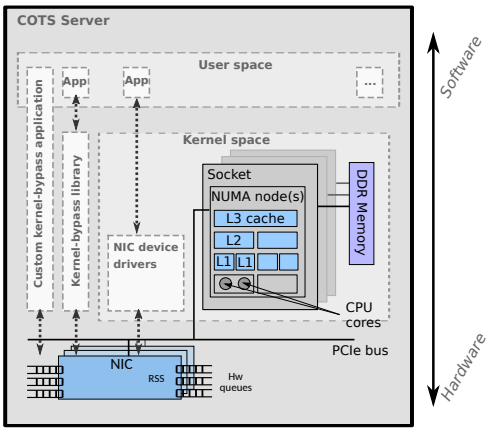

Figure 1: Architecture of a COTS server and the underlying subsystems.

approach is also spreading in other contexts in the form of virtual switches in data center systems, or inter-container routing platforms [19]. In this section we overview the components of COTS hardware, as shown in Figure 1. This will serve as a background to fully understand the acceleration techniques presented in the next section.

2.1 COTS subsystems

The main components of a commodity server hosting network appliances are the *multicore processors* or CPUs, the main memory (aka *RAM*), the *NICs* and the *operating system*.

CPU The Central Processing Units of a server form the subsystem responsible for the execution of the network functions (normally implemented using a programming language such as C or Lua). CPUs are very complex systems, consisting of multiple pipelines with different stages that instructions have to traverse in order to complete the processing. Example of such stages can be the instruction fetch, the execution within the arithmetical-logical unit (ALU), and the memory access to write the result. Furthermore, CPUs include a multi-level cache hierarchy consisting of small but fast memories used to store both instructions and data. Caches optimize the computation time by storing the most recent instructions, or by pre-fetching some data that the program is likely to be accessing: this principle is called *temporal/spatial locality of reference* [20], and it is a very important concept for accelerating packet processing applications [12]. The clock frequency of common CPUs is in the range [1.5, 3.8] GHz[1].

Main memory The memory subsystem is devoted to the data transfer of packets from/to the NICs, as well as the allocation of data structures. The CPU and memory may interact during code execution or data retrieval. Additionally, some off-chip storage units (such as solid-state drives, or SSDs) can be further accessed by the memory subsystem for persistent storage. However, SSDs are out of the scope of this chapter and we omit

[1]See for instance `https://www.amd.com/en/products/servers-processors` or `https://www.intel.com/content/www/us/en/products/processors/xeon.html`.

them. The CPUs being much faster than the RAM memory is a well-known phenomenon, and as a countermeasure, CPUs make extensive usage of smaller, faster caches to reduce the latency needed to access the stored data. This effect is further amplified in modern systems where CPUs are shown to experience a non-uniform latency depending on *where* the data to be accessed are located on the physical chip. This suggested to partition the RAM in different Non-uniform memory access (NUMA) nodes [21], each of them attached to a subset of the available CPUs. As a result, a CPU core can retrieve data from the NUMA node it is attached to much more efficiently than a different NUMA node. Even within a NUMA node, a high level of data locality can significantly reduce the cost of reading from memory: it has been observed, for instance, that sequential data access can keep the cost of reading from the memory predictably small compared to a random access [22]. As we shall see, it is the duty of the high-speed application to make sure that CPUs will always access data located on its own NUMA node, to avoid incurring in performance penalties.

NIC The Network Interface Card is responsible for the I/O of incoming and outgoing packets. NICs are typically connected via a PCIe interface to the host system. Among the most important features of high-speed NICs, we cite the availability of multiple hardware transmit and receive queues, a direct cache access (DCA), and the native support for hosted virtual machines[2]. Hardware queues are essential to achieve horizontal scalability via the usage of multiple CPU per queue: through Receive-side scaling (RSS) [23] the incoming traffic can be distributed to different queues accessed by different cores, each of them receiving only a subset of the input load. A similar mechanism is used on the transmit side, where different CPU cores can write in parallel to separate HW queues without the need of expensive locks. NICs can autonomously write the packet data to a shared memory area without interrupting the CPU (via DMA), and with DCA they can even directly access the last-level cache of a CPU core, which significantly reduce the latency due to the memory write/read operations. Finally, the native support to virtual machines can be leveraged to automatically dispatch traffic to different VMs without the intervention of the hypervisor.

Operating system COTS server are managed by an operating system (OS), which is usually a *nix flavor. The OS's tasks include maintaining low-level services such as memory management or CPU scheduling, and providing an interface to access the hardware resources. A typical network application is a user-space process that is programmed to access the NICs via a either a system call, or a user-space device driver, which in turn are directly connected to the network device. Whereas the original design trend of high-speed network applications was to leverage the kernel-provided driver of the NIC through system calls, it soon became clear that this approach caused a non-negligible overhead for the network function execution. Such overhead grows with the increase of NICs speed, and this inspired the usage of *kernel-bypass* approaches that leverage custom NIC drivers (for example, netmap [24] provides a high-speed interface between a user-space application and the NIC) or a pure user-space implementation of the driver logic, as the Intel DPDK [25] which removes the NICs from the control of the operating system. This trend culminates in the appearance of pure user-space drivers implemented on high-level

[2]The specifications for some popular NICs can be found at the url `https://cdrdv2.intel.com/v1/dl/getContent/331520` (for Intel 10Gbps NICs) or at the url `https://cdrdv2.intel.com/v1/dl/getContent/331520` for Mellanox SmartNICs.

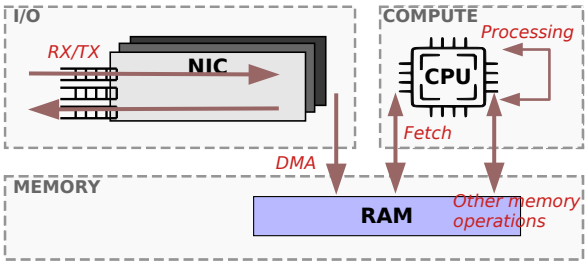

Figure 2: Logical view of the typical workflow of packets in an accelerated software networking application, with the three components of I/O, Compute and Memory.

languages different than C [26]: this approach can provide good performance by inheriting the advantages of kernel-bypass, while at the same time simplifying the development process of custom applications. Though offering significantly high performance, the drawback of kernel-bypass techniques is that they cannot access OS tools such as the Linux implementation of TCP/IP, and they require to rebuild the usual kernel's functionalities at user-space, whereby it is not uncommon to encounter clean-slate (re)implementation of full networking stacks or security mechanisms [27, 28].

2.2 Components of the packet-processing workflow

In order to understand the rationale behind the proposed acceleration techniques, it is useful to focus on two important trends in COTS systems. First, during the last twenty years, the clock frequency of common CPUs has been stagnating, and it nowadays rarely exceeds the 4 GHz [29]. Even though the number of transistors of the microprocessors' architectures kept increasing, this has resulted in an increment of the number of available cores, or in the improvement of the internal caches. One may think of a CPU as a system that provides computational resources in the form of available clock cycles. For an optimized program, the more clock cycles are available, the more operations can be performed. Since the "budget" of clock cycles depends on the CPU clock frequency that has barely changed in the past years, the computational resources of a single CPU core has remained almost constant, offering only slightly better performance than those of old processors[3]. On the other hand, the speed of NICs has steadily increased since the introduction of 10Gbps NICs in 2002 [10], reaching multi-100Gbps capabilities in 2020. Moreover, as COTS servers may have several attachments for multiple NICs on the same chassis, the available bandwidth within a single high-speed COTS server will increase faster than the available computational power. In other words, whereas it is easy to attach additional NICs to the same COTS server, the computational power (represented by the number of available processors) should be considered as constant from purchase time. With this dichotomy in mind, we now analyze the different typologies of resources offered by COTS systems for high-speed network applications: the *I/O*, the *memory* and the *compute*, represented in Fig. 2. An optimal usage of these components is essential to

[3]We will see in the next sections that it is possible to exploit low-level parallelism to speed-up the single-core performance in modern architectures, as also reported in [29]

build high-speed packet processing applications.

I/O The *input/output* (I/O) of a system refers to the pressure of input/output traffic over the entry point of the communication, that is the NIC, during the RX/TX phase of the packet processing. Several aspects may affect the I/O of a system, including the available bandwidth of the NIC or the workflow of the packet processing required by the incoming packets. The latter significantly affects the I/O pressure to the subsequent components of the processing workflow. We consider as an example a forwarding application that receives packets from a NIC port, processes them by swapping the MAC addresses, and sends them to another NIC port. This involves a certain load due to the I/O (the reception and forwarding of the packet) and a subsequent load to the CPU (the MAC address swap). As explained in [24], small packets put a stress on the compute, while large packets affects mostly the I/O and the memory. In fact, if the traffic rate is constant, a workload of small packets translates to a higher packet rate to be processed by the CPU. On the contrary, large packets cause a lower I/O packet rate, which requires less operations per second. For this reason, a typical stress test scenario performed on software switches is executed by sending minimum-sized packets at the maximum I/O speed (e.g. 64-byte packets for a 10Gbps NICs, resulting in a packet rate of 14.88 millions of packets per second). In general, the I/O capabilities of the system are limited by the available bandwidth on the NIC (for large packets) or by the CPU (for small packets), and can be optimized by a load-balancing or batching mechanisms (cfr. Sec 3.1).

Memory After packets are received, the NIC can read/write data from/to some device, which may be another NIC, a RAM memory or a storage unit. This data transfer usually involves the access to the internal bus of the PCIe that connects the NIC to the COTS server. Among the factors affecting the load on the memory component, we include the available bandwidth of the NIC and the bandwidth of the internal buses. For the sake of simplicity, we assume that the PCIe interface has enough capacity to accommodate the requirements of the external NIC. When these assumptions are satisfied, we can assert that the memory capabilities of the system are only limited by the available bandwidth on the PCI bus (typically greater than the maximum capacity of the NIC). The memory efficiency of a system may be improved by reducing the cost of read/write operations, or by issuing memory operations without the involvement of the CPU (cfr. Sec 3.2).

Compute In the context of software networking, we define the *computing power* (or "compute" in short) of a system as the maximum numbers of operations that the system can issue per time unit. The producer of the compute resource is the CPU subsystem. Network applications are the consumers of the compute in the form of packet processing. As a rule of thumb, the CPU frequency is tightly coupled with its computing power: the more clock cycles are available, the more operations can be performed per time unit. However, we must consider that modern CPUs might have several pipelines and multiple levels of caches. This affect the compute in that, with a single clock cycle and optimized pieces of code, it may be possible to issue more than one instruction with a single clock cycle. This value is measured by the *instructions per clock cycle* (IPC), and it reflects the *computing efficiency* of the system (cfr. Sec. 3.3). As a result, it is possible that newer CPUs with several pipelines and a low CPU clock frequency can issue more instructions per time unit than older CPUs with a higher frequency but a single pipeline. The compute phase is a bottleneck (i) when the workload consists of small packets, or

Table 1: Objectives of software acceleration techniques. Each technique may tackle one of three components of the workflow of a packet processing application, namely I/O, Compute and Memory. The objectives are: (1) reduce memory access; (2) optimize memory allocation; (3) share overhead of processing; (4) reduce interrupt pressure; (5) horizontal scaling; (6) exploit CPU cache locality; (7) reduce CPU context switches; (8) fill CPU pipelines; (9) exploit HW computation; (10) simplify thread scheduling, (11) runtime code optimization.

Section	**Technique**	**I/O**	**Compute**	**Memory**	**Objectives**
3.1 RX/TX	Polling	✓	✓		4, 7, 10
	I/O batch	✓		✓	4, 7
3.2 Memory	Zero-copy			✓	1
	Mempools	✓	✓	✓	2
	Hugepages		✓	✓	1, 2
	Prefetching		✓	✓	1
	Cache alignment			✓	1, 2, 6
3.3 Threading	Lock-free multithreading	✓	✓		5, 6, 7
	Lightweight threads	✓	✓		5, 7, 10
3.3 Coding	Compute batch		✓	✓	3, 6, 8
	Multiloop		✓		3, 8
	Branch prediction		✓		8, 11
	JIT compilation		✓		11
3.4 NIC-support	RSS	✓	✓		5
	Flow hashing		✓		9
	SR-IOV		✓	✓	5, 9
3.5 CPU-support	SIMD		✓	✓	8, 9
	DDIO	✓		✓	1, 6, 9
	Hyperthreading		✓		5, 8

(ii) when the processing rate depends on the packet size (as for cryptographic functions, where increasing the packet size may result in increased required instructions per packet).

3 Software acceleration techniques

We now review the software acceleration techniques employed for high-speed packet processing. The approach adopted for accelerating the processing can be focused on *pure software* solutions, involving modifications at every layer that can be modified by coding, or *hardware-assisted*, which takes advantage of common pieces of equipment attached to the COTS server. We list the considered techniques in Table 1, which extends the table presented in [12]. For each row we provide the position of the considered technique when deployed in a typical packet processing workflow (also corresponding to the organization of current section). We also identify the component of the packet processing affected by the acceleration technique between I/O, Compute and Memory (cfr. Sec. 2.2). The last column of the table shows the main objectives of the considered acceleration technique.

3.1 RX/TX

A very important phase in a packet processing application is undoubtedly represented by the reception and the transmission of packets. This phase's main activity is included in the I/O category of the packet processing workflow, and as one can expect the main acceleration techniques in this stage involve some mechanisms to deal with fast I/O. However, as packets may also be treated in an interrupt-driven fashion, interrupt mitigation approaches aim also at reducing the load on the CPU processing, thus targeting the Compute phase of the workflow.

Polling techniques have been proposed long ago [30] to reduce the problem of livelock, that is when a system is overwhelmed by the I/O interrupt rate and the computational resources are used to serve the incoming interrupts rather than perform useful work. When interrupt-driven mode is active, the NIC will send an interrupt signal to the system's CPU to notify that some packets have been received and are ready to be processed. At the reception of such signal, the CPU executes a special interrupt-handling routine (IRQ) causing a corresponding context switch, which may introduce significant performance degradation, especially under heavy loads which could lead to full CPU saturation. Instead, with polling enabled, the CPU periodically queries the device(s) to check whether some I/O to be handled is present, thus removing the need for interrupts. After their first proposition, polling techniques have been successfully introduced to the Linux kernel under the name of NAPI [31] and are implemented by modern user-space drivers such as DPDK [32]. There are some variations on the utilization of polling. For instance, the original Linux NAPI approach proposes to switch between interrupt-mode and polling mode after a first packet is successfully received, turning off the interrupts for all subsequent packets in a batch. Alternatively, with *interrupt-coalescing* [33] the system waits for a fixed timer for further packet reception before raising an interrupt. In general, under heavy loads the optimal solution is achieved by a pure polling mode, in which the CPU continuously polls the device without either of interrupts or timers. This is usually called a *busy-poll* mode. The tradeoff for high-efficiency under heavy load is a 100% CPU consumption regardless of the input rate, resulting in excessive CPU cycles consumption under the average scenario.

I/O batching is another technique aiming at reducing the impact of fast I/O on the system. With I/O batching, the RX/TX is performed on groups of several packets rather than individually. When the NIC receives some packet from a HW queue, it temporarily stores the packet within the queue. It then groups the last received packets in a batch, and moves the batched data at once to a shared memory that is ultimately accessed by the CPU for processing. Differently from interrupt-coalescing (which works on packet reception), the same behavior is also present upon the transmission of packets, whereby the CPU could wait for a batch of packets to fill the TX queue before issuing the transmission operation. Although it can be implemented as an independent mechanism, I/O batching is commonly used in conjunction with polling to mitigate interrupt pressure and at the same time limit the memory operations. The additional advantage is that the transmission of a few bytes on the PCI bus has a very high overhead, because it is required to take ownership of the bus, which is a costly operation. The transfer of multiple packets at once allows a per-transfer PCI access, thus the overhead becomes negligible as the number of bytes being actually transferred increases. I/O batching tackles both the I/O stage as well as the memory stage, by amortizing the cost of I/O and memory oper-

ations over multiple packets [12]. There exists a trade-off between the benefits provided by batching and the per-packet latency introduced to the packet-processing workflow. In fact, increasing the number of batched packets is very effective for small batch sizes, and it has diminishing returns for larger values, while always increasing latency [19]. Therefore, modern libraries and tools for high-speed network applications rarely exceed a maximum batch size of 512 packets (i.e., the default batch values are 32 for DPDK, 512 for netmap). However, the benefits of batching are so high that after being introduced in PacketShader [34] and DoubleClick [35], all major high-speed packet processing frameworks implements it [36, 25, 24, 37].

3.2 Memory optimization

Memory management is another important step for a high-speed network application. Besides the actual reception/transmission of packets (which involve the allocation or deallocation of buffers where the packets are stored) memory can also be accessed by the application itself when needed. Depending on the scenario, there may be two limitations: (i) the memory transfer rate, which can slow down the packet processing by introducing unnecessary latency, (ii) the RAM maximum size, which may cause the system to *swap* some memory to the storage.

Nowadays, the systems employed for high-speed software networking rarely show RAM limitations of the second type, but the overhead linked to the memory transfer rate is still significant. For example, after the correct reception of some packets, the network driver needs to provide the received data to some application: in the past, this was achieved by a costly `memcpy` [38] operation. Via Direct-memory Access (DMA) the NIC can write/read memory areas without the CPU's intervention; together with an explicit mapping of the DMA region to a portion of memory of the network application itself, it is possible to completely avoid memory copy operations. This approach, commonly known as *zero-copy* [39, 24, 25] reduces the memory accesses to a bare minimum, as only lightweight packet descriptors (including a pointer to the packet buffer and some metadata) are transferred between the NIC driver and the network application, while the actual packets are never copied.

Interestingly, most of the existing accelerated frameworks actually avoid performing any kind of memory management at all within the datapath, that is kept as fast as possible by massively adopting *mempools* of preallocated packet buffers. In this way, pools of memory of preassigned size are created and initialized only once at the startup of the application: such pools are used to store packet buffers and are never deallocated. If the NIC has multi RX/TX queue capabilities, it is possible to attach a subset of different HW queues to different memory pools. The NIC can then manage such pools as ring queues, where packets are queued and dequeued in a circular fashion.

Standard *nix operating systems use a virtual memory system that organizes physical memory in pages of 4 kB size by means of an indirection table [40]. The usage of small pages is not suitable at high-speed as many indirections may be required by the application, which degrades the performance of the Translation Lookaside Buffer (TLB), the entity responsible for providing virtual-to-physical memory mappings. Instead, memory is allocated from the Linux *hugepages* with size ranging from 2 MB to 1 GB. This improves the performance by reducing the number of pages to be managed by the TLB by several orders of magnitude, which in turn reduces the possible misses in the TLB table.

Moreover, as discussed in Sec. 2.1, the RAM in modern COTS servers is normally divided into multiple *Non-Uniform Memory Access* (NUMA) nodes, with different CPU cores attached to different NUMA nodes. An important speed-up is obtained when the application has *NUMA-awareness*, that can be achieved by placing the allocated mempools to the same NUMA node where the CPU core is deployed, and accessing other NUMA nodes only when indispensable. In this way, it can be guaranteed that the majority of run-time memory accesses will have a small latency, and only a small fraction may incur on the penalty of accessing a different socket.

In general, memory accesses are significantly faster if *cache-friendly* data structures are adopted. CPUs can read and write data in multiple of a cache-line, and therefore it is essential for the data structure to occupy as few cache lines as possible (which maximizes the CPU cache hit rate) and to be cache-aligned as much as possible (to avoid the transfer of unrelated data belonging to another item which exceeds a cache line). When inter-core transfers are required, cache-friendly data structures can prevent cache-realignment among different CPU cores, hence avoiding not only sharing data among different threads, but also avoiding to access to different data stored on the same cache line. Together with cache-alignment, *prefetching* data while performing some computation can increase the performance of the packet processing by reducing the latency due to memory accesses that are performed ahead of time while the CPU executes different instructions, thus avoiding CPU stalls.

3.3 Threading and coding

As COTS servers are commonly equipped with multi-core chips, software developers have started to leverage different forms of multi-threaded programming, which can reduce the complexity of the I/O, as well as the processing load on CPU cores. Network applications can be well suited for this execution model: if we consider for instance a routing network function, the action taken on a packet does not usually depend on previous or subsequent packets, which suggests that different threads may serve different packets independently. Although a workload that involves stateful computation might invalidate this assumption, this situation can be handled by the application itself. A general guideline advocates to preserve flow-coherence on a per-core basis (such that packets belonging to the same flow are always processed by the same CPU core) and avoid packet reordering. This is feasible thanks to the availability of multiple hardware RX/TX queues, allowing each CPU core to operate on different traffic subsets (see more on Sec. 3.4). On the other hand, it is essential to avoid thread synchronization, which may involve mutual exclusion primitives that can introduce a non-negligible overhead. The alternative is to adopt a *lock-free multi-threading* model, which leverages per-core variables and isolated lock-free data structures. If inevitable, one must privilege lightweight mutual exclusion primitives like read-copy-update as alternative to standard mutexes and semaphores.

The aforementioned execution model is called *run-to-completion execution*: in this workflow, each packet is treated by a separate thread (which will also process other packets of the same flow) and the processing is terminated by the final forwarding decision. Alternatively, frameworks such as DPDK have recently introduced a *cooperative multi-threading* model [41]. This model adopts *lightweight threads* (called `lthreads` in DPDK's lingo) whose execution depends on a simple cooperative scheduler less complex than the classical POSIX `pthread` scheduler [42] (for instance, the `lthread`-scheduler lacks thread

prioritization and preemption). In a cooperative model, the entire packet processing workflow can be split into multiple subsets, each of one managed by a different thread. Therefore, a thread implements only a portion of the processing activity and can then voluntarily issue a context-switch to another thread for further processing.

In combination with the different multithreading approaches, some coding best-practices can be used to significantly speed-up the software execution and improve the system's scalability. The main idea is to explicitly write the code in some particular fashion in order to "help" the generation of low-level instructions and thus maximizing the throughput of the underlying CPU. For example, with *compute batching* it is possible to improve the CPU pipeline performance [11, 36] by writing code to explicitly process packets in large batches rather than on a per-packet basis. This technique extends the I/O batch to the inner computing path, and it mitigates the overhead of network function processing. Software network engines that use compute batching either adopt the same batch size previously used for the I/O batching, or they opt for some multiple of the max batch size for the I/O. In addition, compute batching can increase the usage efficiency of the L1 instruction cache, as the processing to be performed on a batch is likely to be the same for all packets in a batch: in this way, as soon as the first packet enters the computing phase, the code to be executed is fetched and loaded into the L1 instruction cache (which may generate some overhead), but for all subsequent packets in the same batch the code would be already in the cache, thus speeding-up the processing for the rest of the batch. Compute batching also helps filling the internal CPU pipelines which can work at full capacity: as the instructions to be executed on different packets are typically independent, it can be possible to execute multiple instructions within the same clock cycle [36], thus enhancing the IPC value. Finally, compute batching can be effectively used in conjunction with memory optimizations such as data prefetching (cfr. Sec. 3.2) and can take advantage of CPU-assisted primitives to improve data-level parallelism (cfr. Sec. 3.5).

Another coding best-practice is the *multi-loop* programming, an evolution of the classical *loop unrolling*, which consists in writing packet processing loops to explicitly process packets in groups of two or four in each iteration (hence, the conventional name of dual-loop or quad-loop respectively). One of the advantages of loop unrolling is to reduce the number of `jump` instructions which may cause a CPU pipeline invalidation: at the end of a loop, the processor typically "jumps" to execute an instruction that is not adjacent to the jump instruction, and this can introduce idle pipeline slots because of the fetching of new instructions and the refill of the CPU pipeline. At the same time, multi-loop can be effectively used in conjunction with prefecthing data from subsequent packets while processing already prefetched packets, which can maximize the usage of the CPU pipeline [36].

Regardless of the particular threading model chosen, high-speed network applications should avoid context switches as much as possible, which be caused by a system call, or by the OS scheduler to allow the execution of another thread. Since a context-switch requires to store the current state of a process and replace it with another process, it has a strong impact on the processing flows, as the low-level pipelines are invalidated and need to be repopulated with different instructions. Even simple function calls, though not causing a full context-switch, can show a non-negligible performance penalty, as the running thread needs to push/pop local variables from the stack, invalidate the CPU pipelines and load new instructions from a potentially random memory location. As a consequence, developers frequently use explicit *inlining*, which requires the code annota-

tion of frequently accessed functions whose code is duplicated and copied every time that it is used (as opposed to raising a jump instruction towards the memory area containing the function's code). This technique reduces the run-time overhead of frequent function calls, at the cost of a slightly increased program text size [43]. Sometimes code jumps may be unavoidable, i.e. when the result of some `if` condition produces a code branching. In this case, with workload-related insights it may be possible to guess in advance if one particular branch can be taken with high probability. The programmer can then manually annotate the code to indicate the most likely execution branch corresponding to a conditional expression, thus forcing the CPU to follow a specific branch which may improve the CPU's *branch prediction* success rate. A correct branch prediction allows the CPU to continue filling its pipelines without waiting for the actual result of the condition (which normally stalls the pipeline by introducing so-called "bubbles" to wait for the result to be computed [20]). In contrast, a branch misprediction causes a full pipeline invalidation and the subsequent repopulation with the instructions related to the correct branch. This implies that to avoid incurring in performance penalty, it is of extreme importance to avoid branch mispredictions.

Although the majority of the presented coding practices assume the usage of a compiled language such as C, there exist some high-speed network applications written in interpreted programming languages (as for Snabb [44], programmed in Lua [45]). The typical preference of compiled low-level languages (and C in particular) is because of performance: code compiled into machine language and optimized for execution may run faster than interpreted code whose translation is usually on a per-line basis. Moreover, high level languages might show unpredictable behavior, e.g. in the case of garbage collection, that is usually interleaved with the regular program execution [46]. On the other hand, the ahead-of-time compilation may perform static optimization which translates the code into low-level instructions without any assumption on the run-time execution. With *Just-in-Time* (JIT) compilation, it is possible to get the best of the two worlds. A JIT compiler performs an intermediate step that analyzes the code to be executed and optimizes it at run-time. Among the advantages of this approach, the compiler can access run-time information such as the values of variables, or the parameters of a function call (not present during a static compilation step), which can improve the execution speed. Furthermore, with JIT compilation it is possible to optimize the code that is actually used more frequently, as opposed to a static compilation that targets a global optimization. Finally, interpreted code may be more portable than compiled code as it would not require a new compilation for a new target machine [47].

3.4 NIC-assisted acceleration

Modern NICs are equipped with HW components that can be accessed by software to provide an advanced set of functionalities. This category of *NIC-assisted acceleration* techniques can reduce the I/O pressure or relief the computation load on the CPU.

One of such NIC components is a specialized device accessed to natively compute flow hashes that can be exported via the network driver to the high-level application. This is useful if the application involves the usage of hash-tables, as the CPU will not need to recalculate a new hash value, which may be computationally intensive. The availability of multiple hardware RX/TX packet queues is leveraged by *Receive Side Scaling* (RSS) [23], a technique that enables the adoption of a per-core multithreading approach: the NIC

uses the pre-calculated hash value (computed over the 5-tuple to preserve flow coherence) to distribute packets among different CPU cores, thus achieving flow-level parallelism. Since packets belonging to the same flow will always be scheduled to the same CPU core, this also improves the locality of data structures on a per-core basis. Furthermore, the RSS seed can be configured such that packets of bidirectional transport connections are processed on the same CPU core.

In addition to these advantages, modern NICs can even provide virtualization support by offering native communication facilities between packets received by or sent to a virtual machine hosted in the same COTS system equipped with the NIC. It is the case of *Single root input/output virtualization* (SR-IOV), which allows NIC's PCI resources to be logically accessed as separate virtual channels, each of them used by a different virtual machine, without being explicitly managed by some hypervisor. Thanks to SR-IOV enabled NICs, several virtual functions can independently access the NIC's resources without explicitly manage concurrent access: this improves the system's scalability in a virtualized environment, though the maximum number of concurrent VMs is limited by the hardware specs.

3.5 CPU-assisted acceleration

The last class of techniques analyzed in this chapter are *CPU-assisted acceleration* techniques, which leverage the features of modern CPUs to accelerate the I/O, compute and memory management of packet processing applications.

As observed in Sec. 2.1, most modern CPUs are equipped with multiple pipelines and computing elements (ALUs) that can support low-level data-parallelism through *Single Instruction Multiple Data* (SIMD) operations. When using SIMD, it is possible to allow simultaneous computation on multiple data while issuing just a single instruction. The effect of SIMD is maximized when the same instruction has to be performed on different objects (e.g. same instructions on different packets of the same typology). By allowing the execution of a single instruction on multiple data instances, this approach can improve the throughput of the CPU pipelines, especially when using a batched processing approach.

Furthermore, modern CPUs and NICs can effectively interact by taking advantage of *Data Direct I/O* (DDIO), which is an improvement over the usual DMA that allows NICs to perform a direct transfer of packets into the last-level CPU cache (typically the L3 cache) instead of the main memory. By avoiding multiple reads from and writes to the main memory, DDIO additionally improves the cache data availability, as the possible misses on the L1-L2 caches would eventually result in a hit on the L3 cache with high probability. Finally, CPU chipsets are nowadays capable of *hyperthreading* (HT), a technique that lets a single CPU core appear as two independent virtual cores for concurrent scheduling of multiple processes per physical core. The goal of HT is to share the CPU resources of a single physical core to multiple threads that may be concurrently scheduled to the different virtual cores of the same physical core, thus increasing the number of independent instructions in the pipeline. Hyperthreading plays also an important role to tackle memory latency, as while a (HT) thread waits for a memory operation, the other virtual core can perform some useful operations. As such, multiple instructions can operate on separate data in parallel, which can increase the IPC, and the efficiency of the pipeline is enhanced as a result of a reduced CPU idle time.

4 Summary

As network softwarization steadily proceeds, more and more network appliances are challenging the classical design choice of relying on custom purpose-specific equipment, in lieu of a more flexible approach based on pure software deployed on COTS servers and commodity NICs. Acceleration techniques are an effective mechanism for leveraging the capabilities of such systems and keep up with the ever increasing requirements of complex network applications. Due to their versatility, acceleration techniques tackle the performance bottlenecks of the packet processing workflow and reduce the overhead required for I/O, compute and memory operators. More generally, there exist several classes of engines which exploit one (or more) of the acceleration techniques to provide low level building blocks for accelerated applications (as for netmap [24], DPDK [25] or the ixy drivers [37]), purpose-specific functions (i.e., traffic generation [48], access control lists [49], load-balancing [50]) or fully-fledged modular frameworks for accelerated packet processing (such as FastClick [11], the Snabb switch [44], OpenVSwitch [51]).

In this environment, network operators can decide to deploy their appliances as virtual network functions to be executed in production servers, incurring in a performance penalty that has been continuously reduced for the last decades. State-of-the art software switches can easily achieve multi-10Gbps performance even with a single dedicated CPU core (as shown in [52], most of the analyzed frameworks can process more than 10 millions of packets per second, with peaks of 50 Gbps depending on the packet size). The performance gap between HW and SW solutions is definitely outmatched by the advantages in flexibility, ease of management and scalability brought by the latter. The evolution of the available acceleration techniques has fueled the proliferation of an ecosystem of software networking engines that can be used to implement high-speed packet processing under a plethora of scenarios, ranging from network monitoring, to firewall systems or even fully-fledged NFV environments [52]. We conclude this chapter by providing some insights on the exploration of network applications' design space, an important yet often underrated step which should be done in order to carefully choose one or more acceleration techniques depending on the desired objectives.

Kernel-bypass vs in-kernel approaches. High-speed packet processing is not necessarily a kernel-bypass exclusive field. Despite most of existing solutions have adopted a pure user-space approach based on DPDK [11, 19, 44], there are some alternatives which rely on the Linux kernel and, at the same time, provide enhanced performance. One popular representative of the in-kernel software network engines is the eXpress Data Path (XDP) [53]. xDP operates directly in the Linux kernel at driver-level, by providing a fast interface between the NIC and an extended Berkeley Packet Filter (eBPF) program [54]. The eBPF subsystem is a virtual machine deployed inside the Linux kernel that can be used to map some user-space program to be executed following the reception of a packet. A programmer can write some code that is translated into eBPF instructions that are in turn verified by the eBPF verifier in order to prevent integrity or security problems (cfr. the dedicated chapter on eBPF). We underline that an XDP/eBPF application can process almost 20 millions of packets per second on a single 3.6 GHz CPU core (only slightly worse than a pure DPDK counterpart) and can saturate the whole PCI bus with just 4 cores. More importantly, the xDP/eBPF approach does not require a full stack implementation at user-space, as it is integrated within the kernel and it can reuse all the

available codebase without reinventing the wheel. Furthermore, the security and integrity checks performed by the eBPF are a plus to ensure that the deployed program does not break the host system. Overall, network developers should decide whether to adopt one approach or the other, depending on the target use-case and the security/performance requirements.

Regular NICs vs SmartNICs. In this chapter we mostly focused on ordinary NICs and COTS equipment. However, *SmartNICs* [6] have become quite popular in a very diverse set of use-cases, including energy saving [55], DDoS mitigation [56] or acceleration in the public cloud [57]. The definition of a SmartNIC relates to a design in which a regular NIC integrates within the ordinary chipset an additional hardware-programmable element, such as FPGAs or a System-on-chip [12]. The typical advantage of SmartNICs over regular NICs is the additional computational power deployed on-chip that can be used to further offload the COTS server's CPU by taking additional charge. This can be taken to the extreme case, where a full control-plane plus the corresponding data-plane can be both deployed on the computing element of the SmartNIC [6]. Finally, for latency-sensitive applications, the SmartNIC can provide better performance guarantees. The main drawback relies on the circuitry complexity, which may increase the management overhead and decrease the time-to-upgrade an existing network appliance. As for the previous design choice, it is up to the developer to analyze the desired use-case and opt for the best-suitable solution.

References

[1] Anirudh Sivaraman, Thomas Mason, Aurojit Panda, Ravi Netravali, and Sai Anirudh Kondaveeti. Network architecture in the age of programmability. *ACM SIGCOMM CCR*, 50(1):38–44, 2020.

[2] Juniper Networks. Openflow support on juniper network devices. `"https://www.juniper.net/documentation/en_US/release-independent/junos/topics/reference/general/junos-sdn-openflow-supported-platforms.html"`, visited on 01/06/2020.

[3] Pica8. Fully automated whole-campus open networks that are a snap to deploy. `"https://www.pica8.com/product/"`, visited on 01/06/2020.

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR*, 44(3):87–95, 2014.

[5] Barefoot Networks. Tofino switch. `"https://www.barefootnetworks.com/products/brief-tofino/"`, visited on 01/06/2020.

[6] Netronome. What makes a nic a smartnic, and why is it needed? `https://www.netronome.com/blog/what-makes-a-nic-a-smartnic-and-why-is-it-needed/`, visited on 29/06/2020.

[7] Alex Galis, Stuart Clayman, Lefteris Mamatas, Javier Rubio Loyola, Antonio Manzalini, Slawomir Kuklinski, Joan Serrat, and Theodore Zahariadis. Softwarization of

future networks and services-programmable enabled networks as next generation software defined networks. In *IEEE SDN for Future Networks and Services (SDN4FNS)*, pages 1–7, 2013.

[8] Edward Jones. Cloud market share – a look at the cloud ecosystem in 2020. `https://kinsta.com/blog/cloud-market-share/`, visited on 29/06/2020.

[9] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and Frans Kaashoek. The Click Modular Router. *Operating Systems Review*, 34(5):217–231, 1999.

[10] The Ethernet alliance. The Ethernet 2020 Roadmap. `https://ethernetalliance.org/technology/2020-roadmap/`, visited on 01/11/2020.

[11] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In *ACM/IEEE ANCS*, page 5–16, 2015.

[12] L. Linguaglossa, S. Lange, S. Pontarelli, G. Rétvári, D. Rossi, T. Zinner, R. Bifulco, M. Jarschel, and G. Bianchi. Survey of performance acceleration techniques for network function virtualization. *Proceedings of the IEEE*, 107(4):746–764, 2019.

[13] A. Galis, S. Clayman, L. Mamatas, J. Rubio Loyola, A. Manzalini, S. Kuklinski, J. Serrat, and T. Zahariadis. Softwarization of future networks and services -programmable enabled networks as next generation software defined networks. In *2013 IEEE SDN for Future Networks and Services (SDN4FNS)*, pages 1–7, 2013.

[14] Intel. Intel 64 architecture. `https://www.intel.com/content/www/us/en/architecture-and-technology/microarchitecture/intel-64-architecture-general.html`, visited on 01/06/2020.

[15] AMD. Amd64 architecture programmer's manual. `https://www.amd.com/system/files/TechDocs/24594.pdf`, visited on 01/06/2020.

[16] ARM. Arm instruction set. `https://iitd-plos.github.io/col718/ref/arm-instructionset.pdf`, visited on 01/06/2020.

[17] Bruno Chatras and François Frédéric Ozog. Network functions virtualization: The portability challenge. *IEEE Network*, 30(4):4–8, 2016.

[18] Jinho Hwang, K K_ Ramakrishnan, and Timothy Wood. NetVM: high performance and flexible networking using virtualization on commodity platforms. *IEEE Transactions on Network and Service Management*, 12(1):34–47, 2015.

[19] Dave Barach, Leonardo Linguaglossa, Damjan Marion, Pierre Pfister, Salvatore Pontarelli, and Dario Rossi. High-speed Software Data Plane via Vectorized Packet Processing. *IEEE Communication Magazine*, 56(12):97–103, 2018.

[20] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[21] Christoph Lameter. NUMA (non-uniform memory access): An overview. *ACM Queue*, 11(7):40–51, 2013.

[22] Ulrich Drepper. Memory part 2: CPU caches - Figure 3.15: Sequential vs Random Read. `https://lwn.net/Articles/252125/`, visited on 01/10/2020.

[23] Tom Herbert and Willem de Bruijn. Scaling in the linux networking stack. `https://www.kernel.org/doc/Documentation/networking/scaling.txt`, 2011.

[24] Luigi Rizzo. netmap: a novel framework for fast packet I/O. In *USENIX ATC*, pages 101–112, 2012.

[25] Intel. Data plane development kit. `http://dpdk.org`, visited on 01/07/2020.

[26] Paul Emmerich, Simon Ellmann, Fabian Bonk, Alex Egger, Esaú García Sánchez-Torija, Thomas Günzel, Sebastian Di Luzio, Alexandru Obada, Maximilian Stadlmeier, Sebastian Voit, and Georg Carle. The Case for Writing Network Drivers in High-Level Programming Languages. In *ACM/IEEE ANCS*, pages 1–13, 2019.

[27] EunYoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *USENIX NSDI*, pages 489–502, 2014.

[28] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP acceleration as an OS service. *Proceedings of the 14th EuroSys Conference*, pages 1–16, 2019.

[29] Karl Rupp. Microprocessor trend data. `https://github.com/karlrupp/microprocessor-trend-data`, last updated on 15/02/2018.

[30] Jeffrey C Mogul and KK Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[31] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. Beyond softnet. In *Annual Linux Showcase & Conference*, volume 5, pages 18–18, 2001.

[32] Intel. Dpdk poll mode drivers. `https://doc.dpdk.org/guides/prog_guide/poll_mode_drv.html`, visited on 15/06/2020.

[33] 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*, 24(6):1118–1128, 2013.

[34] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: a GPU-accelerated software router. *ACM SIGCOMM CCR*, 40(4):195–206, 2010.

[35] Joongi Kim, Seonggu Huh, Keon Jang, KyoungSoo Park, and Sue Moon. The power of batching in the click modular router. In *Asia-Pacific Workshop on Systems*, 2012.

[36] Leonardo Linguaglossa, Dario Rossi, Salvatore Pontarelli, Dave Barach, Damjan Marjon, and Pierre Pfister. High-speed data plane and network functions virtualization by vectorizing packet processing. *Computer Networks*, 149:187 – 199, 2019.

[37] Paul Emmerich, Maximilian Pudelko, Simon Bauer, Stefan Huber, Thomas Zwickl, and Georg Carle. User space network drivers. In *2019 ACM/IEEE ANCS*, pages 1–12. IEEE, 2019.

[38] Linux man pages. `https://man7.org/linux/man-pages/man3/memcpy.3.html`, visited on 05/07/2020.

[39] Deri, L. PF_RING-ZC. `https://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy`, visited on 05/07/2020.

[40] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. O'Reilly Media, Inc., 2005.

[41] Intel. The l-thread subsistem. `https://doc.dpdk.org/guides/sample_app_ug/performance_thread.html?highlight=lthread#lthread-subsystem`, last updated on 15/06/2020.

[42] Dick Buttlar, Bradford Nichols, Dick Buttlar, Jacqueline Farrell, and Jackie Farrell. *Pthreads programming: A POSIX standard for better multiprocessing*. O'Reilly Media, Inc., 1996.

[43] 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 Conference*, pages 539–552, 2016.

[44] Michele Paolino, Nikolay Nikolaev, Jeremy Fanguede, and Daniel Raho. SnabbSwitch user space virtual switch benchmark and performance optimization for NFV. In *IEEE NFV-SDN*, pages 86–92, 2015.

[45] Lua. Lua – the programming language. `https://www.lua.org/`, visited on 10-06-2020.

[46] Alastair Reid, John McCorquodale, Jason Baker, Wilson Hsieh, and Joseph Zachary. The need for predictable garbage collection. In *Workshop on Compiler Support for System Software*, 1999.

[47] John Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35(2):97–113, 2003.

[48] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 2015 Internet Measurement Conference*, pages 275–287, 2015.

[49] J. Daly, V. Bruschi, L. Linguaglossa, S. Pontarelli, D. Rossi, J. Tollet, E. Torng, and A. Yourtchenko. Tuplemerge: Fast software packet processing for online packet classification. *IEEE/ACM Transactions on Networking*, 27(4):1417–1431, 2019.

[50] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q Maguire Jr, Panagiotis Papadimitratos, and Marco Chiesa. A high-speed load-balancer design with guaranteed per-connection-consistency. In *USENIX NSDI*, pages 667–683, 2020.

[51] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, Awake Networks, and Martín Casado. The Design and Implementation of Open vSwitch. In *USENIX NSDI*, pages 117–130, 2015.

[52] Tianzhu Zhang, Leonardo Linguaglossa, Massimo Gallo, Paolo Giaccone, Luigi Iannone, and James Roberts. Comparing the performance of state-of-the-art software switches for NFV. In *ACM CoNEXT*, page 68–81, 2019.

[53] The Linux Foundation. eXpress Data Path (XDP). `https://www.iovisor.org/technology/xdp`, visited on 01/07/2020.

[54] Linux Socket Filtering aka Berkeley Packet Filter (BPF). `https://www.kernel.org/doc/Documentation/networking/filter.txt`, visited on 01/07/2020.

[55] Karthikeyan Sabhanatarajan, Ann Gordon-Ross, Mark Oden, Mukund Navada, and Alan George. Smart-nics: Power proxying for reduced power consumption in network edge devices. In *IEEE Computer Society Annual Symposium on VLSI*, pages 75–80, 2008.

[56] Sebastiano Miano, Roberto Doriguzzi-Corin, Fulvio Risso, Domenico Siracusa, and Raffaele Sommese. Introducing smartnics in server-based data plane processing: The ddos mitigation use case. *IEEE Access*, 7:107161–107170, 2019.

[57] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: Smartnics in the public cloud. In *USENIX NSDI*, pages 51–66, 2018.