Computer Communications 196 (2022) 148–166

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

Review

A survey on accelerating technologies for fast network packet processing in Linux environments

Eduardo Freitas *, Assis T. de Oliveira Filho, Pedro R.X. do Carmo, Djamel Sadok, Judith Kelner

Networking and Telecommunications Research Group (GPRT), Federal University of Pernambuco, Brazil

ARTICLE INFO

Keywords:
Fast packet processing
Linux kernel
Kernel bypass
Offloading
Programmable network

ABSTRACT

The path a packet takes when handled by the Linux Kernel has been well established for a long time. Its overhead/bottleneck issues are also known. Nonetheless, complexity has increased with the introduction of new paradigms such as virtual machines, containers, software-defined networks, multicore CPU architectures, etc. Coupled with the need for low delay and high bandwidth services, multiple technologies emerged in an attempt to solve these issues, generally known as Fast Packet Processing Frameworks. However, each technology provides a better network processing in different ways, using different methods and solutions, yielding to different benefits and trade-offs. There is a need to shed light not only on the processing overhead of standard Linux processing, but also highlight the sometimes overlapping and confusing Fast Packet Processing solutions. In this work, we propose a taxonomy to classify these current solutions into the three main groups corresponding to their targeted architecture, (i) hardware, (ii) software, (iii) and virtualization. We detail how each solution works and then discuss its applicability in real-world scenarios according to four different criteria: (i) host resource usage, (ii) high packet rate, (iii) system security and (iv) flexibility/expandability. Followed by a discussion of real use-cases that combine and compare multiple solutions discussed in the paper.

Contents

* Corresponding author.
E-mail addresses: eduardo.freitas@gprt.ufpe.br (E. Freitas), assis.tiago@gprt.ufpe.br (A.T. de Oliveira Filho), pedro.carmo@gprt.ufpe.br (P.R.X. do Carmo), jamel@gprt.ufpe.br (D. Sadok), jk@gprt.ufpe.br (J. Kelner).
URL: https://www.gprt.ufpe.br (E. Freitas).

https://doi.org/10.1016/j.comcom.2022.10.003
Received 28 December 2021; Received in revised form 21 September 2022; Accepted 5 October 2022
Available online 11 October 2022

1. Introduction

In recent years, Internet services have become ubiquitous to people's social and business activities, in addition to conquering new domains such as power distribution, e-health, and banking systems [1]. With such success, there is a need for more processing power and higher communication resources. As a result, high-throughput and low-latency traffic processing is at the heart of the design of new technologies such as 5G, 6G, time-sensitive networks (TSN) for industry and entertainment, and software-defined programmable networks (SDN). To keep up with the stringent new requirements, high-speed network cards with speeds of 40 and 100 Gbps are commonly deployed in data center servers and switches [2–4].

Existing programmable networks and data center infrastructure is based upon the Linux Kernel. Being a general-purpose kernel, it is flexible and supports multiple protocols and drivers. User applications, including web access, E-mail exchange, and file transfer, execute in the user space. The kernel controls shared networking code related to the network and transport layers for security and performance reasons. This paper looks at the processing overhead a network packet incurs when handled by the network and transport layer protocols moving between kernel and user space. Our motivation is the new bandwidth and low delay requirements and the impact of the insertion of new layers of code to handle virtual machines and containers, the virtualization of network drivers, the use of hypervisors, and programmable network interfaces. A full-blown kernel network stack also relies on costly packet copying among the different packet buffering areas and operating system mode-switching between user and privileged kernel modes.

The growing demand for enhanced packet processing on systems that use the Linux Kernel led to the creation of different special-purpose technologies ranging from software frameworks to programmable hardware devices to cope with such limitations. For example, in 2010, Intel introduced the Data Plane Development Kit (DPDK) [5] packet processing optimization mechanism. Next, mTCP, a highly scalable user-level TCP Stack for Multicore Systems (2012) [6] and the fast packet I/O framework netmap (2014) [7], were also proposed. More recently, Programmable NICs gained more widespread adoption [8–11]. Due to the diversity of the suggested approaches to solving the packet processing bottleneck, we realized the need for a study highlighting their inner working details and discuss their benefits, strengths, and weaknesses in different scenarios.

This article initially describes how packets are processed in the Linux Kernel, from their arrival at the network interface to when they move to the application running in the user space. We then detail and clarify all the issues involved in this process to help the reader better understand how each proposed technology positions itself to improve packet processing. In addition, we conduct a discussion about the different packet processing technologies presented in this work. To achieve this, we consider some criteria, such as (i) host resource usage, (ii) high packet rate, (iii) system security, and (iv) flexibility/expandability while seeking to highlight and add to the elucidation of technologies. Finally, we also present a concise explanation of some consolidated combinations of performance optimization technologies for packet processing, in order to help the reader understand how all these technologies are used in real-world scenarios. To the best of our knowledge, this article is the first to provide a deep review that brings together recent advances in packet processing from multiple varieties. We hope the reader will find it helpful to demystify some concepts, help make technology choices, and point the way forward.

The article is organized as follows: in Section 2, we describe the packet processing path followed inside the Linux Kernel and highlight its potential bottlenecks. In Section 3, we provide an overview of existing technologies' concepts and categories, proposing a taxonomy to organize and group each of them. We classify them into three main groups, (i) hardware, (ii) software, (iii) and virtualization, detailed in Sections 4–6, respectively. In Section 7 we choose four criteria based on real-world demands to analyze and discuss each solution group, explaining how each attends a given criterion. Finally, Section 8 concludes the review and points to some general future directions.

2. Packet processing in Linux

This section introduces the reader to the way packets are processed in the Linux kernel [12] and the overall path that a packet traverses, along with the issues that this processing brings, starting at the Network Interface Card (NIC) and moving to reach the application in user space. Then, it follows with a discussion of current native solutions that Linux already provides to enhance packet processing performance.

2.1. Standard packet processing

We illustrate the standard packet processing in Fig. 1 to ease understanding of this process.

To increase performance at the Linux kernel level, a NIC manages packets through the circular queue of buffers known as *ring buffers*. See the "NIC" area (colored green) in Fig. 1. Each buffer is referenced through a descriptor and occupies a slot in the ring buffer, containing the address and length of its memory space [7]. Initially, this buffer is an empty `sk_buff struct` data structure that is pre-allocated when the ring buffer is initialized and set to the "ready" state. Given that the NIC device driver is responsible for maintaining the ring buffer, the `sk_buff` buffer is accessible by the NIC through the driver only [13].

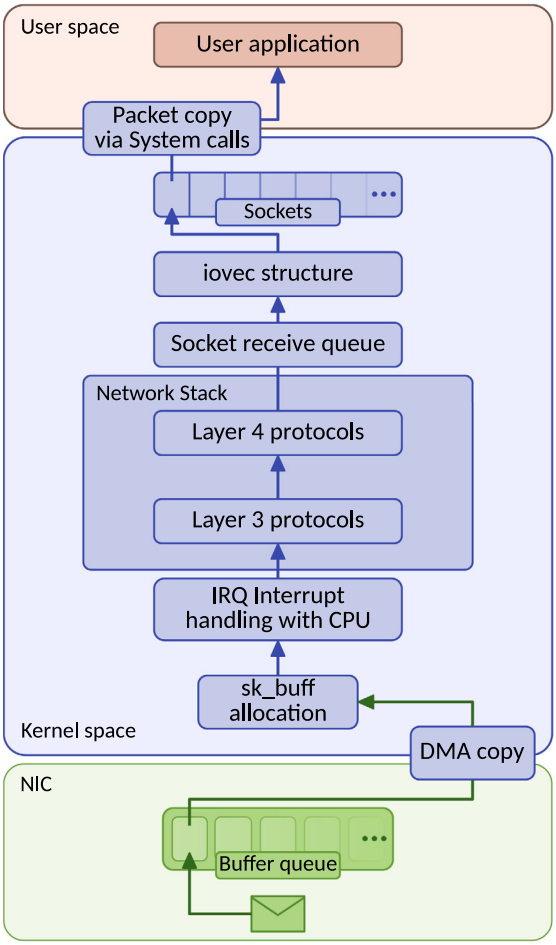

Fig. 1. Network packet path from NIC to user space in the Linux kernel.

As soon as a packet arrives at the NIC coming from the network, it is copied through Direct Memory Access (DMA) [14] to the next available `sk_buff` structure, marking the respective buffer descriptor in the ring buffer as "used". See the "Kernel space" area (colored blue) in 1. It is the first copy process the packet goes through. Then the NIC sends a hardware interrupt request (IRQ) to notify the CPU, telling it to handle this new packet. The CPU responds to the IRQ by executing a registered Interrupt Service Routine (ISR). This routine first disables any further interrupt requests from the NIC's receive interrupts. Then, the CPU core polls the network device that originated the interrupt to get the received packet from the `sk_buff` into its specific First-In, First-Out (FIFO) queue called poll list or poll queue [15]. The ISR ends by raising a soft interrupt (*softIRQ*) to take the packet processing from interrupt context to process context. The function `net_rx_action()`, in *softIRQ* context, starts processing the packet and takes it up to the kernel stack to start specific protocols processing [16].

The kernel then processes layer three protocols, such as IP, performing integrity or packet checksum verification, applying rules, and deciding where to forward this packet to or passing it to higher protocols for additional processing. In additional processing, transport layer protocols such as TCP [17] are invoked. After possible packet reassembly due to packet fragmentation by the network or transport layer, the data is then passed to the user space application. First, it is sent to the receive queue of the correspondent socket through `sock_queue_rcv_skb()`. Next, any process that previously signaled a system call to receive packets from that specific socket (e.g., `recv()`) is notified, and the data is copied again to an `iovec` struct. Then, this packet is sent to user space via the receive system call where the application process is running [18].

Despite the apparent simplicity of the Linux packet processing path, some key components and steps became a performance bottleneck issue. This is mainly apparent when current data centers demand high throughput and low latency traffic. We explain these main issues next and summarize them in Table 1.

- **sk_buff allocation**: The kernel uses the `sk_buff` data structure to hold packets. To begin with, this structure can be considerably large, primarily because it carries information about multiple protocols and their headers. This becomes a bigger problem for applications that do not need such protocol headers to be allocated in memory. Furthermore, the frequent **sk_buff** allocation and reallocation cause CPU stress. Studies conducted in [19] show that initializing `sk_buff` buffers spends approximately 5% of the CPU cycles used in the packet reception process. Managing allocation and reallocation consumes around 8%, and controlling requests for these memory operations use 50% of the CPU cycles.
- **NIC Interrupt Processing**: As explained, the NIC interrupts the CPU to initiate packet processing, which triggers the kernel to receive the packet and allocate its `sk_buff`. With a growing processing load, the *irqbalance* daemon, responsible for managing interrupt triggers, switches to "performance mode", where it spreads the interrupts to all CPUs available to avoid throttling one CPU core. This causes user applications to be frequently interrupted by incoming packets that are creating interrupts in multiple cores [20].
- **System calls**: These calls also are responsible for packet processing delay and performance loss. System calls require a constant context switch between user and kernel modes. This context switch is expensive in terms of CPU time [21]. Studies conducted in [22] show that the context switch can add about 39µs of CPU overhead when processing workloads that still fit the CPU cache. If the workload does not include the CPU cache, this overhead can increase to around 203µs.
  Furthermore, the context switch caused by system calls requires the CPU to change between user to kernel-mode state. The CPU has different structures like L1, L2, and L3 memory caches and Translation Look-aside Buffers (TLB). When the processor changes between states, it also populates all of its structures with this state, replacing the previous state with information for the new one. This replacement is called *processor state pollution* [23], and is responsible for performance degradation, causing evicted cache entries, for example, or wasted CPU cycles dedicated to enabling state switching.
- **Sockets**: The socket API also contributes to the overhead problem when processing packets within Linux. The socket API does not support a concurrent scenario, where an application will read and write simultaneously to a socket, for example. When performing operations such as socket reading, the application will likely block and stay suspended until the operations finish [24]. Some applications work around this issue by creating a multi-process scenario for each connection or a multi-thread scenario. The problem with this workaround is introducing more overhead because they require thread switching and event synchronization. Also, sockets require frequent system calls since applications run in user space and a socket operates in kernel space. This switching of modes reduces performance, as indicated previously.
- **Memory copies**. As earlier explained, a packet goes through at least two copying processes inside the kernel. These operations also increase CPU processing and resource consumption, raising an additional performance bottleneck [25]. Studies carried out in [26] show that depending on the buffer size, it is possible to save up to 39% of system cycles when transmitting packets using mechanisms that avoid copying buffers.

Table 1
Standard Linux processing overhead summary.

Overhead	Issues
sk_buff allocation	Memory consumption, CPU cycles
NIC IRQs	User space applications interruption
System calls	CPU cycles, processor state pollution
Sockets	Single-operation sockets, frequent system calls
Memory copies	Memory consumption, CPU cycles
Network Stack	"*unnecessary*" processing
Unitary processing	CPU cycles

- **Network stack**. Linux is a general-purpose kernel, so handling layer 3 and 4 protocols are mandatory. However, specific applications may consider the network stack a performance bottleneck issue [6,27,28], especially those that do not need such layers or do not require the use of their full-featured implementation.
- **Unitary packet processing**. Some Linux kernel drivers can batch packets from NIC to **sk_buff** allocation. However, the kernel processes next to each packet singularly [29]. This means that the kernel processes each packet at a time [7], so all the copying processes, CPU cycles, and system calls issued repeat for every packet received/sent. This translates to higher considerable overhead, especially in a cloud computing environment with a high data transfer rate among servers and the presence of numerous small packets arriving constantly.

2.2. Native Linux packet processing solutions

As explained, Linux poses multiple performance issues regarding the fast packet processing scope. Because of this, Linux started to implement native solutions for different purposes, each attempting to enhance packet processing in specific scenarios. We will explain the main ones next.

- **NAPI.** The *New API* (NAPI) came in the 2.5.7 Linux version [30] and refined the way network cards interacted with the kernel functions and the system. It removed unnecessary interrupts, added support for early hardware packet drop, and introduced a fair trade-off between throughput and CPU usage. Currently, NAPI is the standard packet reception mechanism for every packet that traverses Linux, including the interrupt handling explained in Section 2.1. But, an important characteristic of NAPI is that the IRQ-driven packet reception explained only happens at low network loads [15]. If the system reaches 100% utilization, or the incoming network load grows faster than it can process, NAPI changes to a poll-driven mechanism, stops using interrupt and starts polling packets directly from the NIC. This improves throughput in turn of higher CPU utilization since polling naturally consumes more CPU [15,31]. However, if packets would continue to be processed in an IRQ-driven manner, the CPU would also throttle faster and not process any incoming new packets.
- **RSS.** Receive Side Scaling (RSS) [32] is a hardware capability that enables a multi-queue network card to distribute incoming packets among its queues to different CPU cores and thus processes them in parallel. A special feature in this setting is the possibility of pining a CPU to a specific queue, enabling one CPU to process packets only from one queue, and enabling a multi-core system to process multiple packets in multiple CPUs in parallel. The condition for which packet goes to which queue can be a hash function over the network and/or transport layer headers [32], for example, IP address, transport protocol, and port. The hash is used with a redirection table to determine which CPU will process the packet. This should be used in conjunction with the Extended Message-Signaled Interrupts (MSI-X), a different type of interrupts a device can handle. MSI-X can direct interrupts to specific CPU cores. Furthermore, MSI-X provides vectors of interrupts that allow handling interrupts simultaneously by multiple CPU cores. This allows configuring RSS to not only pin each queue to one specific CPU core but also pin every interrupt from that queue to be handled by the same CPU core [33], useful to avoid reloading the CPU cache.
- **RPS.** Receive Packet Steering (RPS) is a software correspondent of RSS [32]. It is useful when there is no hardware with RSS compatibility, or the desired protocols cannot be parsed in hardware RSS. The RPS driver calculates a hash from the packet's header field to determine which CPU to send the packet to, similar to the filter that RSS calculates. Then, it places the allocated network packet's `sk_buff` on the chosen CPU backlog queue and interrupts the CPU to wake up and process that packet [34]. Then, the CPU handles the packet, processing it through the network stack.
  Also, there is a constraint when using RPS. As the filter to steer packets is based on protocol headers, if the chosen protocols and headers represent a single connection flow that dominates the incoming traffic, one CPU core will overload, and performance may suffer. This may be an indication of the misconfiguration of chosen filters.
- **RFS.** Receive Flow Steering (RFS) [32] is the "next step" of RPS. It considers the network application locality as the filter to steer packets into multiple CPU cores. Suppose a network application runs in CPU core A. In that case, it is ideal that the incoming packets for this application are also inserted in the queue of the same core, which optimizes cache usage. RFS uses the same mechanism to filter packets as RPS, placing packets into the CPU backlog and issuing interrupts to wake up the CPU.

3. Classification of fast packet processing approaches

Depending on the objective under consideration, packet processing solutions can be diverse. Some are more efficient in freeing host resources to competing tasks other than networking, whereas others are more concerned with achieving higher packet processing rates. Sometimes, a trade-off between the two goals must be considered when configuring data center servers. Given that not all packet processing techniques are drop-in solutions, administrators, and even developers need to understand the different mechanisms to apply them correctly to their use case.

We created a taxonomy to help categorize such diversity of packet processing solutions and illustrate it in Fig. 2. To achieve this, we divided existing solutions following two levels of criteria: **architectural** and **technological**. The architectural criteria gather the solutions according to the type of adopted architecture that serves the fast packet processing. Initially, we observed that each solution targets the context of a specific given system *architecture*. Hence, a given solution often requires to be deployed in a specific and different architectural environment. We identify the following broad architectural classes: (i) hardware, (ii) software, and (iii) virtualization. This work classifies the surveyed packet processing solutions into one of these three architectural environments.

Then, inside these three classes, we separate each solution based on the different types of technique used to achieve high packet processing rates. This refers to the adopted implementation and, more importantly, the strategy used to enhance packet processing.

As explained, the first architectural criterion is Hardware. We divided it into TCP Offload Engines, Programmable NICs, RDMA NICs, and GPU Packet Processing. These technologies rely on moving packet processing directly to hardware, each using a different approach to do so.

The second architectural criterion is Software. This category is divided into Zero-copy Networking, Kernel Bypass, In-Kernel Processing and User Space TCP techniques. These approaches rely on software implementations to drive packet processing.

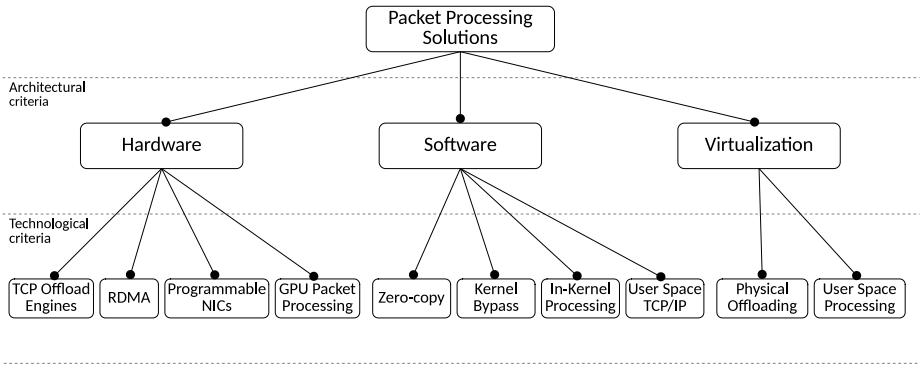

Fig. 2. Classification of fast packet processing solutions based on the architectural and technological criteria.

The last architectural criterion is Virtualization. It describes different approaches used to enhance the connection of a physical NIC to the virtual ones inside the VMs. Each technology in this class develops its own distinct strategy to speed up packet processing while usually establishing a trade-off between performance and flexibility. The virtualization class can be divided into the Physical Offloading and User Space Processing techniques.

4. Hardware solutions

Specialized hardware is often used to accelerate packet processing by moving processing tasks from the host to such specialized hardware. An example is the offload of functions from the kernel and general-purpose CPUs onto dedicated hardware. There are four classes of hardware solutions that differ in the level of functions they support and the type of hardware they offload to. Namely, **TCP Offload Engines**, **RDMA NICs**, **Programmable NICs** and **GPU Packet Processing**, explained next.

4.1. TCP Offload Engines

The work in [28] describes an example of an Offload implementation. It builds specialized hardware, called TOE (TCP Offload Engine), that implements the TCP/IP protocols stack.

As such boards implement the entire lower four protocol layers, they enable the operating system to bypass the network stack. This reduces CPU resource usage and avoids overhead from increased CPU cycles, excessive memory copies, and system calls bottleneck.

Nonetheless, implementing TOEs often comes with risks and high costs [35]. The complexity of implementing such extensive protocols in silicon makes it expensive and raises deployment issues like making changes to the hardware on large-scale infrastructure. Also, finding and fixing bugs is more complex, especially in the case of non-programmable TOEs.

Examples of implementations of TOEs can be found in [36], achieving around 9 Gbps throughput with a basic implementation of TCP that supports re-transmission and the reordering of duplicated segments. The work in [37] implements a complete Gigabit TCP/IP TOE in an FPGA circuit, with support for MAC, ARP, ICMP, IP, TCP, and also UDP. However, this Gigabit Ethernet implementation achieved a throughput limited to only 949Mbps in the conducted experiments. AccelTCP [38] performs TCP offload operations such as connection setup and teardown entirely to the NIC. FlexTOE [39] shows how to decompose the TCP data-path into a fine-grained data-parallel pipeline to support full and flexible offload to on-path NPU-SmartNICs. Fig. 3 illustrates the general TOEs structure. More specifically, it is possible to observe the structure of the TOE, in the "TOE" area (colored green) in Fig. 3.

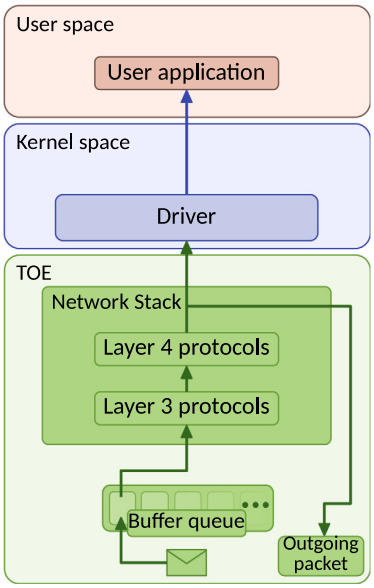

Fig. 3. TOE's structure inside the Host.

4.1.1. Simple functions offload

There are also other types of Offload Engines, or more simply Offload NICs, with hardware that implements processing and enables the offloading of a single function of the TCP/IP stack, like IP checksum calculation or other tasks like hardware timestamping. The Intel 82576EB 1GbE NIC [40] is an example of a simple NIC that supports checksum offload.

Also, another type of NIC Offload is the Segmentation Offloads that some NICs implement, which has native Linux support. They are usually divided into TCP Segmentation Offload, UDP Fragmentation Offload, Generic Segmentation Offload, Generic Receive Offload [41].

In principle, LSO (Large Segment Offload), or Large Send Offload, is a mechanism inserted in network cards that seek to fragment large data packets and reduce CPU usage. The TSO (TCP Segmentation Offload) feature is used if the processed packets are only TCP packets. On the other hand, if the incoming packets are UDP packets, the UFO (UDP Fragmentation Offload) mechanism is used.

The purpose of TSO is to provide the TCP segmentation processing inside the physical network card. Thus, if a large block of data sent by an application does not exceed a particular value, that block will be sent directly through the Linux network stack to the network card driver queue. Then, TCP segmentation is performed corresponding to the maximum segment size (MSS) provided by skb. As the TSO engine is a hardware feature, it requires not only a physical implementation in the NIC, but also the Linux kernel driver support. Furthermore, it

is also necessary to support partial checksum offload usage and to be able to aggregate multiple physically disjoint memory regions in a single data transfer, the so-called Scatter–Gather operation. Cheng and Cardwell [42] discuss changes in the implementation of TCP on Linux, including the TSO for the high-performance packet scheduler.

The UFO, as indicated above, has the same operating principle as the TSO. However, as the UDP protocol layer alone does not have extensive datagram segmentation functionality as in TCP, the segmentation process will occur at the network layer (IP segmentation) when the length exceeds the MTU. Thus, UFO makes it possible for the Linux kernel networking stack to assign IP segmentation functionality of large UDP datagrams to the hardware. Thereby reducing stack overhead when fragmenting large UDP Datagram to MTU-sized packets. Unlike TSO, UFO is currently obsolete, and only older kernels or TunTap devices (or similar) via pull-through still support UFO skb generation.

The mentioned technologies relate to the sending process, but receiving packets can also count with some offloading capabilities. When the NIC receives fragmented packets, LRO (Large Receive Offload) can help to automatically combine into a more significant chunk of data and send it to the operating system for processing at once.

However, hardware that supports these technologies is not always available. In these cases, GSO (or Generic Segmentation Offload) is another alternative that seeks to optimize packet processing inside Linux. Despite being an isolated mechanism, the GSO can be used with the TSO or even replace the UFO. Regarding the UFO, when using GSO, the Linux UDP protocol stack provides UDP segmentation logic instead of the IP segmentation used by the UFO, which makes it possible for each segment to contain the entire UDP header and allows the continuation of the GSO segmentation at the IP layer. GSO evaluates if the NIC supports TSO before sending it to the card in combination with TSO. If the NIC supports TSO, the packet is segmented. Otherwise, the protocol stack is divided into segments and delivered to the driver. The authors of [43] discuss the use of GSO to optimize UDP content delivery.

4.2. RDMA - Remote Direct Memory Access

There are NICs that offer Remote Direct Memory Access (RDMA). This technology allows the moving of data from the memory of a host to another one over the network without the participation of the operating system. By offloading this operation to hardware, RDMA reduces latency and CPU usage. Existing approaches offload transport to RDMA NICs [44,45]. The work in [46] describes SockDirect, a high-performance RDMA-based socket system that makes use of RDMA. Designed as a user space high-performance socket system, SockDirect leverages both RDMA and shared memory (SHM) technologies for inter-host and intra-host communication, respectively. The authors claim to achieve a latency close to the one achieved by the underlying SHM queue and RDMA raw technologies. More specifically, the results highlight 0.3 µs RTT latency for intra-host socket communication corresponding to the small fraction 1/35 of Linux socket and only 0.05 µs higher than a bare-metal SHM queue. Similarly, a 1.7 µs low RTT between RDMA hosts was reported, almost the same as the duration of a raw RDMA write operation, and corresponds to 1/17 of Linux latency. Fig. 4 illustrates a RDMA host system communication.

The authors report its remarkable scalability with the number of cores about throughput. During the experiments, a single thread could send 23 (i.e., 20 times that of Linux) and 18 Million messages per second (i.e., 15 times that of Linux and 1.4 times more capacity than raw RDMA write operation) for intra-host and inter-host communications, respectively. The proposed solution removes costly operations such as multi-thread synchronization, buffer management, large payload copy, and process wake-up overheads from the data plane.

Nonetheless, the authors single out some limitations to SockDirect. Among them is a performance loss due to frequent cache miss events when encountered in simultaneous connections (to thousands). Other problems such as TCP packet head-of-the-line blocking, congestion spreading, and deadlocks may emerge as most RDMA NICs implement a Priority-based Flow Control (PFC).

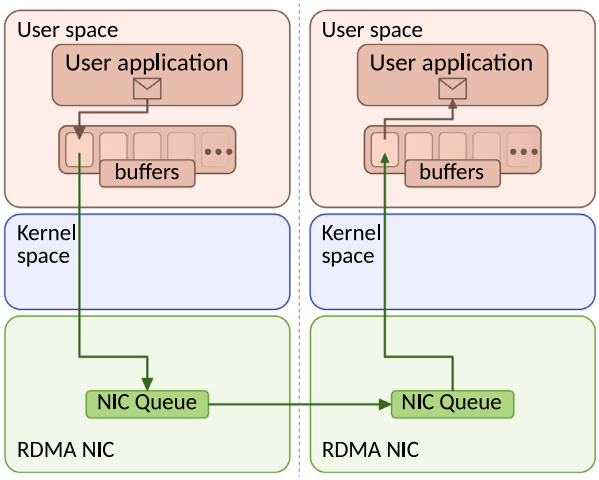

Fig. 4. Illustration of a RDMA system with two hosts.

4.3. Programmable NICs

This category of solutions uses technologies composed of NICs with greater processing power. Programmable NICs, also called SmartNICs, differ from common NICs since they support the processing of network packets inside the network card itself, thanks to their available processing hardware. Programmable NICs, also referred to as Data Processing Units (DPUs), are similar to the TOEs in the sense of also offloading specific packet processing tasks from the host to ease its resource usage. However, instead of having such functions implemented in hardware, it delivers a programmable one, enabling the design of network functions and protocols on demand. Examples of existing programmable NICs implement functions such as packet filtering, Distributed Denial of Services (DDoS) mitigation [9] or even generic flow-level network functions [47].

Implementing Programmable NICs mainly involves FPGA-based hardware or may take place directly on SoC (system-on-a-chip) devices. The latter emerged in data center environments, with multiple companies developing their Programmable NICs products like Mellanox, Broadcom, Huawei, and Netronome [8]. Most SoC Programmable NICs consist of a **computing unit** that contains a general-purpose ARM/MIPS multi-core processor and specific hardware accelerators, including deep packet inspection, encryption/decryption, hashing, and compression functions. A NIC also supports **DMA access** to allow communication with the host. Lastly, it comprises an **on-board memory** and a **traffic control module** that manages TX/RX ports and packet buffers, responsible for delivering packets to the suitable Programmable NICs cores. The work in [48] discusses and evaluates the benefits and performance limitations of offloading a networking application to NICs.

There are different programming and accessing packets on Programmable NICs, as determined by the NIC manufacturer. Some cards require the programmer to write native hardware and DMA commands. In contrast, others leverage a complete operating system in their processors, allowing communication with the Linux stack, DPDK or XDP applications. We discuss these technologies later in this paper.

4.4. GPU packet processing

While initially designed as a specialized processor for the acceleration of graphics rendering, GPUs have been used in different data processing tasks such as machine learning, gaming applications, and more recently in the handling of network packets, see [19,49–51].

PacketShader [19], APUNet [49], GASPP [50], and GPU-Ether [51] leverage both GPU high parallel processing capacity and large memory

bandwidth. Most works use GPUs to reduce CPU packet processing overhead, achieve high throughput, and lower packet latency. For instance, at the heart of the design of PacketShader [19] are the elimination of the overhead emanating from single and multiple packets based memory management as well as the offloading of packet processing tasks to the GPUs. The solution demonstrates that by prioritizing the parallel processing of multiple operations, the NVIDIA GTX480 GPU reaches a peak processing performance equivalent to that of 10 x5550 processors. However, when the number of packets for processing is low, the CPU offers better performance. This result is explained as the CPU prioritizes the execution of a single task instead of the GPU.

A second contribution that makes use of GPUs for packet processing is APUNet [49]. The processor manufacturer Advanced Micro Devices (AMD) developed the AMD Accelerated Processing Unit (APU) in the last decade. This is a single 64-bit microprocessor chip designed to act as a central processing unit (CPU) and graphics processing unit [52]. The APUNet solution demonstrates the use of the APU GPU to increase the performance of packet-intensive processing algorithms such as those used for the real-time encryption of network traffic. Furthermore, the authors argued that the reason behind CPUs outperforming GPUs, in general, is due to the presence of a bottleneck in the Peripheral Component Interconnect Express (PCIe) used for communication between a CPU and the GPU dedicated for packet processing. As a result, they advocate for the use of APU as it integrates CPUs with GPUs and hence removes the above-mentioned bottleneck. Despite this, the integrated GPUs cannot benefit from the special GPU memory and its high bandwidth Graphics Double Data Rate (GDDR). This is a memory specialized for fast rendering on GPUs. As a result, the authors decided to use a zero-copy technique when processing packets.

A third GPU-related piece of research dedicated to building high-speed packet processing applications on GPUs is GASPP [50]. Here, the authors argue that there are multiple significant issues with GPUs in the context of packet processing. They cite the lack of programming abstractions and GPU-based libraries suitable for processing network traffic. As a result, a great deal of effort is often required for building even the simplest of applications, such as packet decoding and filtering. In addition, they mention that the CPU still executes the complex critical-path operations, such as flow tracking and TCP stream reassembly, hence, losing GPU gains. The authors also consider the drawback identified in APUNet, namely, the cost of copying data between a NIC to user space and from there to the kernel space and then to the GPU. A programmer would spend a considerable effort to optimize such data transfer. According to its authors, GASSP manages the state of TCP flows and their reassembly within the GPU and implements load balancing among the GPU threads while opting for a zero-copy operation between CPU and GPU. As a result, it achieves data rates in multi-gigabit, even for demanding packet processing applications such as intrusion detection, packet encryption, and traffic classification. The authors claim that when testing the same application, they achieved up to a 16.2 times reduction in execution time and speedup compared to standalone GPU-based configurations. Despite the promising results, GASPP may suffer from high packet processing latency due to the batch processing nature of GPUs. In other words, GASPP is not tailored to lead with applications that rely on a per-packet basis processing such as intrusion detection or firewall functions. GPU-Ether [51] is another GPU-based packet processing framework. It allows direct network access to the GPU. It simplifies it by eliminating the complex multi-staged pipe-lining and substituting costly memory transfer operations by peer-to-peer DMA (P2P-DMA) [53] communication. The goal is to reduce packet processing latency and avoid CPU interference. The authors promise that their framework offers a simple GPU programming model. They claim that their solution has effectively developed IPV4 forwarding, IPSec gateway, and network intrusion detection systems.

5. Software solutions

Whereas hardware-based solutions mainly target the use of offloading to free up CPU resources and speed up packet processing, host-level software also offers a considerable performance impact, as discussed in [27]. Due to the operating system overhead issue raised in Section 2, some works addressed this problem from different perspectives. Overall, software architectural designs include strategies such as **Zero-copy Networking**, **Kernel Bypass**, **In-Kernel Processing** and **User Space TCP**.

5.1. Zero-copy Networking

The number of copies a packet goes through during NIC/Kernel operations reduces performance, as earlier stated. Buffer-related read and write operations use as much as 60% of the total CPU cycles used for packet processing-related activities, as shown in the optimization section of [19]. Alongside CPU processing, memory allocation is also hindered since each buffer copying operation results in more buffer space allocation, especially as the `sk_buff` data structure is considerably large since it holds information about the network stack protocols and their headers. "Zero-copy networking" stipulates that whenever a packet is allocated its memory space for the first time – whether for transmitting or receiving – it will remain at that single memory region throughout the whole network operations inside the kernel, avoiding copies especially to/from kernel space and from/to user space.

The work in [26] describes a native implementation of `MSG_ZEROCOPY` for packet sending inside Linux. It introduces a new socket call flag that forces the kernel to lock the buffer into memory using page pinning. This way, the kernel pins the user pages and allocates to a packet data a `sk_buff` structure created from these pages, avoiding packet copying from the user space program to the kernel. It also builds a notification interface to let the manipulating program packets know when the buffer can be reused or updated. For example, the socket error queue receives a notification message containing a timestamp for each packet with an incremental counter and a copy of the original packet to associate the timestamps with the send request. The benchmark carried out in [26] shows a 39% saving of the overall CPU system cycles. It also shows that this mechanism benefits more when processing large-size packets.

Nonetheless, as discussed in [54], the results of zero-copy algorithms do not consistently achieve a satisfactory benefit. One of the reasons is that the technique does not fit every packet processing case and offsets the benefits, such as when dealing with small-sized packets. This is due to the overhead needed to set up zero-copy being significant and not influential when there is small data to copy. Also, `MSG_ZEROCOPY` is available only for transmission operations, meaning that packet receiving – an important scenario when dealing with cloud computing scenarios, for example – must still use ordinary coping mechanisms. Because of this, frameworks and libraries committed to implementing a high packet processing solution decided to combine the usage of zero-copy mechanisms along with the other processing mechanisms. Such software-based solutions include netmap, XDP, DPDK, or IX which we will detail next.

5.2. Kernel Bypass

As the kernel performs a considerable number of operations for each packet that traverses it, it becomes a processing bottleneck. Because of this, removing the kernel altogether from the processing path became an attractive idea that gained widespread adoption in different implementations of new packet processing paths. This method is called "Kernel Bypass", or sometimes "user-level packet I/O". It consists of effectively bypassing the kernel or at least the network stack and exposing the NIC to user space directly. Note that this technique potentially raises security and design issues, addressed differently by each framework we present next.

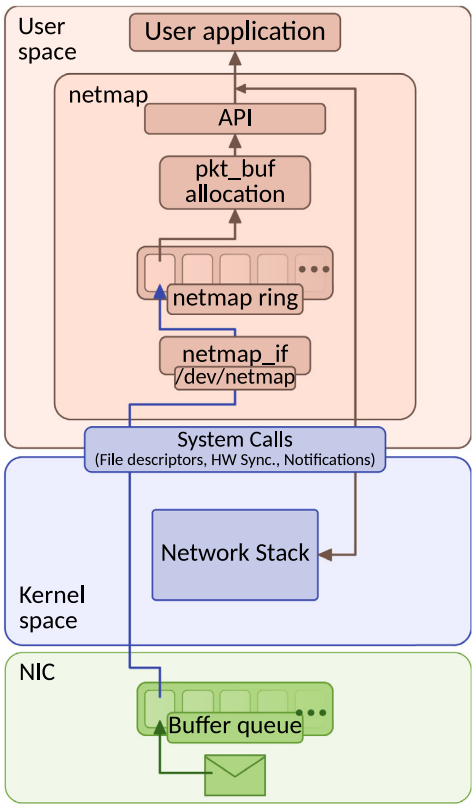

Fig. 5. Netmap's structure inside the Host. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

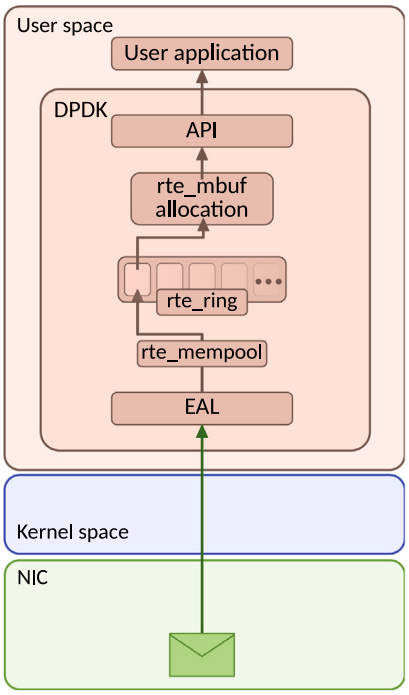

Fig. 6. DPDK's structure inside the Host. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

5.2.1. *netmap*

The `netmap` framework [7] performs a Kernel Bypass that works jointly with the operating system: it combines the creation of new buffer rings that connect to the NIC, and the usage of system calls for event notification and I/O synchronization. Also, it leverages the use of zero-copy technology. The netmap framework modifies some kernel components and introduces patches to the NIC driver. Fig. 5 illustrates netmap's overview structure.

Enabling netmap on a NIC creates a data structure with three objects: `pkt_buf`, `netmap_ring` and `netmap_if`. The kernel allocates these structures in the same user-accessible memory region for all netmap interfaces. This facilitates zero-copy forwarding to user space and also between different netmap interfaces.

The `pkt_buf` data structure has a fixed size and is previously allocated during netmap initialization, avoiding the per-packet overhead of allocating it each time a new packet arrives. The NIC ring buffer and the netmap ring reference each of these packet buffers. The `netmap_ring` is a replica of the ring buffer implemented by the NIC's driver, mentioned in Section 2. It stores information about each packet buffer and information about the ring itself (e.g., total number of slots and number of available slots). Finally, `netmap_if` is the structure that holds information about the netmap interface. All elements previous description can be seen in the "User space" (color pink) area in Fig. 5.

To enable setting up an interface in netmap mode, netmap creates a special device as in `/dev/netmap` and uses the file descriptor of this device to enable the usage of system calls through it, hence it supports the basic I/O control (e.g., using system calls such as `ioctl()`, `select()`, `poll()`). To use the netmap interface, an application uses system calls with the device's file descriptor to initiate reception or transfer of packets, I/O synchronization, and memory access.

In the network processing path described in Section 2, netmap sits on top of the kernel's processing chain, allowing the NIC to communicate with netmaps's ring buffer and allocate its packet buffer. It is responsible for delivering packets to the user space application. Observe that the kernel is not entirely dismissed, as system calls and drivers are still used. For a better interpretation, see Figs. 1 and 5 again.

Some works in the literature demonstrate the use of netmap [55–57]. An interesting take on netmap is seen in [58], where the authors create a virtual switch for VMs in a virtual environment using netmap as API for packet processing and delivery to the hypervisor. This is promising since there is no need for special hardware features and compatibility with the kernel.

5.2.2. *DPDK*

The Data Plane Development Kit (DPDK) [5] represents a set of libraries and drivers that also performs Kernel Bypass to enable high packet processing. Unlike netmap, DPDK does not use traditional system calls. Instead, it relies on a *custom user space API* that applications should interact with. Previous works have performed experiments and built real applications that use DPDK to enhance packet processing [59–61]. The main structure of the DPDK is designed around three of these core components or library: `rte_ring`, `rte_mempool` and `rte_mbuf` [62]. We illustrate DPDK's structure in Fig. 6.

The library `rte_ring` component is the ring buffer structure. It provides packet FIFO queue management, storing objects in a table with a fixed maximum size. It supports batch enqueue and dequeue operations, which provide faster performance by processing multiple packets simultaneously. The `rte_mempool` is the memory pool management structure in charge of allocating objects in memory, referenced in a ring structure. `rte_mbuf` manages the buffers used to store network data. The library `rte_mempool` uses the buffers created by `rte_mbuf` to allocate network data (objects) and reference these objects in a memory pool. See the "User space" (or pink) area in Fig. 6.

DPDK uses a special driver that maps the device's memory region into user space [27]. When a packet arrives at the NIC, the packet is initially allocated in user space memory-mapped, especially to the NIC.

Then DPDK uses the `mbuf` structure to handle the network packet itself. Afterwards it stores the objects using `mempool` to correctly allocate objects in the `ring` table.

As DPDK uses its device driver, this means that it interacts directly with the NIC and receives packets directly from it. This brings a side effect that the network card is no longer available to the rest of the operating system, remaining inaccessible to ordinary system programs like tcpdump, ethtool, or tc. In other words, when DPDK uses a NIC, this becomes no longer visible to the operating system of the host. This breaks system compatibility and forces applications to re-implement functionalities implemented by said programs, and also isolates applications that use DPDK from the rest of the Operating System (OS) [63].

In the OS processing path, DPDK bypasses the kernel, using a user space API to expose the packets from NIC to the application. Unlike netmap, DPDK does not rely on the kernel's costly system calls, implementing its I/O synchronization and memory management.

DPDK implements a Poll Mode Driver that uses busy-polling to access the network card to retrieve, process, and deliver the network packets to the user space application without the need for any interrupts. This also brings the side effect that the CPU usage is always close to 100% in each core used regardless of the network load, which can directly affect performance on other processes. Furthermore, DPDK is not suitable for implementing network services that handle considerably large flows, such as a VPN service, where traffic may be encapsulated in a single sizable flow. Here, a single large flow may overload a given CPU in a server even if it has multiple CPUs and a high capacity link, causing it to collapse.

5.2.3. PF_RING ZC

PF_RING [64] designs a framework with two basic components, a kernel module that implements a low-level packet copying to custom ring buffers and a user space Software Development Kit (SDK) that enables the development of user applications and the interaction of these applications with PF_RING through its API. Fig. 7 illustrates the overall PF_RING structure.

PF_RING divides its structure into different modules. First, it uses the standard Linux network drivers to capture packets from the NIC. The next component is a kernel module responsible for copying the packets from the network driver to the PF_RING memory ring buffer, previously allocated at creation time. This ring buffer acts like the kernel's standard one, giving buffers for packets in a circular queue of descriptors. Still, instead of the kernel, the ring buffer accesses the buffers to receive the packets and bypasses the kernel stack. At the outbound interface and above the ring buffer sits the library `pf_ring`, the user space library component that interacts with the user application and with the kernel module, sending the received packets to the application through the PF_RING API.

The PF_RING framework relies on two other particular components for packet processing. First, the Stack Injection module allows sending certain packets to the kernel network stack for standard processing. This same module receives the packet back, and the application should use the same API to receive these packets. The other component is the ZC (Zero-Copy) module, which implements a complete kernel bypass and zero-copy processing, bypassing even the standard PF_RING module with the ring buffer and standard Linux drivers. This process uses a specific ZC driver in library `pf_ring` (user space) that connects directly to the NIC and exposes packets to the application using PF_RING API. See the "User space" (or pink) area in Fig. 7.

Even though the "Vanilla" PF_RING is an open-source product, the ZC component is not, requiring the user to acquire a license to use the complete kernel bypass approach. Also, PF_RING requires external kernel modules, which hinders deployment on multiple machines, and it also needs updating with the kernel functions and properties.

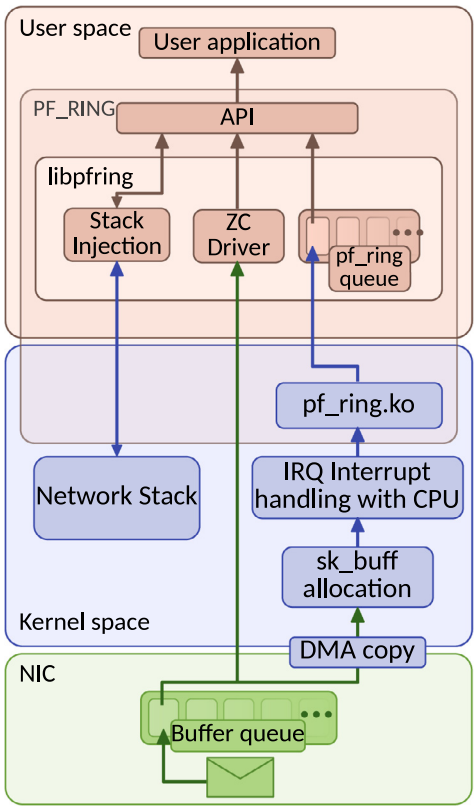

Fig. 7. PF_Ring's structure inside the Host. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

5.3. In-kernel processing

As seen previously, some solutions create a fast user space environment to move some or all processing from the kernel into this user space environment. However, the opposite way is also available, creating a fast processing **inside** the kernel, allowing in-kernel processing. This section studies the most widespread technology in this category.

5.3.1. XDP

Kernel bypass approaches raise some issues. Due to not using the kernel functions, some require re-implementation, whether by the proposed framework itself or the user application. Examples of crucial missing kernel functions include routing tables, protocol stacks, and device drivers. Also, discarding the kernel from packet processing raises security issues. The kernel is responsible for multiple security and isolation measures to protect the system, offering secure services such as a protected memory region, process isolation, and controlled hardware I/O. Also, user applications using kernel bypass solutions end up isolated from the rest of the operating system due to the need for direct hardware access and processing packets in specific user space API. As bypass solutions take complete control of the network card, other existing applications on the rest of the OS are unable to access the NIC, losing compatibility with the OS. The eXpress Data Path (XDP) effort [63,65] emerged in an attempt to deal with these problems, allowing fast packet processing but keeping kernel security and compatibility. Researchers and developers claimed for a long time that a solution to the Linux packet processing bottleneck had to bypass the kernel level and avoid its complexity, XDP inventors set a goal to refute such claim.

To understand how XDP works, one must understand the basics of the extended Berkeley Packet Filter (eBPF). Although the name suggests a filtering role, it is currently generic enough to suit other non-networking cases. Networking software usually requires the constant

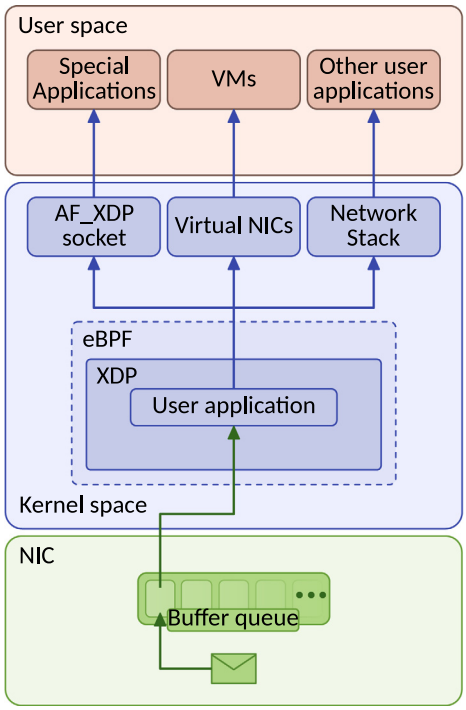

Fig. 8. XDP's structure inside the OS.

changing of kernel source code, increased layers of abstraction, or loading kernel modules. Deemed unpractical, eBPF avoids such mess by running sandboxed programs in the Linux kernel without changing kernel source code or loading kernel modules. This makes the Linux kernel programmable, allowing the addition of more intelligence and new features without the need to introduce more layers of complexity at the risk of lowering efficiency or compromising safety.

Overall, eBPF is an infrastructure inside the kernel that provides a register-based Virtual Machine [66]. This infrastructure supports a *restricted set* of C code that compiles into native machine code running inside the Virtual Machine (VM). See the Kernel space (or blue) area in 8. Additionally, it provides an in-kernel verifier that analyzes the program during compilation to enforce security measures and protect the kernel, checking issues like out-of-bounds memory access or the presence of *never-ending* programs (uncontrolled loops). XDP takes advantage of this infrastructure and builds its libraries inside eBPF. This means that every XDP program executes inside an eBPF VM. Fig. 8 presents an illustration of XDP's structure.

XDP places itself at an early point in the kernel packet processing path: the device driver. This means that driver modification is necessary, although changes are minimal. When a packet arrives, the XDP Driver Hook triggers an eBPF kernel event, and the corresponding eBPF program receives the packet. This takes place even before `sk_buff` structure allocation, so the program receives packets directly from the NIC.

The program can parse packet data, manipulate it, write at any part of the packet, and expand/shrink the packet buffer to add/remove headers [63]. The program can access kernel network functionalities through helper functions exposed by the kernel during the eBPF execution, like the routing table or specific stack processing. Observe that this is achievable without the need to go through the entire network stack.

Alongside this standard packet processing, XDP supports the offloading of applications to a *programmable NICs*, reviewed in Section 4.3. In other words, XDP can send the user application directly to a specific NIC, and it will run inside the network card. However, this feature requires a special configuration. For example, it cannot require kernel helper functions or specific kernel stack facilities. This is necessary because the program needs to operate outside the host. This way XDP acts as an enabler to send the application to the network interface through its device driver, allowing XDP applications to take advantage of using a programmable NIC and its features.

After packet parsing and manipulation, the program finishes its execution with a final decision determining what course of action XDP should take about a packet. This verdict expands the XDP framework even further and is expressed by the following simple return code:

- **Drop**, this will make XDP drop the packet;
- **Re-transmit**, this causes the packet to be re-transmitted out on the same network interface it came from;
- **Pass to Stack**, this will send the packet to the normal network stack path that it would normally go through;
- **Redirect**, the redirect code permits redirecting the packet to three main locations:
  - Network interface: re-transmitting the packet to a different interface from what it came from. This includes virtual interfaces from virtual machines in user space;
  - CPU: continue the processing on another CPU, a decision that is useful for balancing the load on the host;
  - User space: redirecting the packet directly to user space through the new socket family, `AF_XDP`.[1] This option performs kernel bypass, as the kernel stack is not used along with the traditional memory management. The `AF_XDP` socket also uses a zero-copy approach to deliver packets to user space and enhance performance.

The interesting applicability of XDP is the possibility to act as a code accelerator, allowing typical user space applications to run specific parts inside XDP and other parts in ordinary user space. This is only possible due to its compatibility with the operating system, as it runs inside the kernel. An example of such an approach is the BPF Memcached Cache (BMC) [67]. It creates a Memcached application inside XDP that filters incoming web requests and serves Memcached caches for specific requests. When the incoming request is not available in BMC's caches, it sends it to the normal user space Memcached application that will ultimately locate the cache response.

Note that it is possible to find multiple projects in the industry that use XDP and eBPF. Among them, there are Cloudfare [68], Facebook [69], Netflix [70] and Netronome [71].

5.4. User space TCP/IP

The Linux TCP/IP stack has become a considerable bottleneck due to the increasing network capacity and demand for low delay services. The stack has a considerable programming complexity since it is implemented in the kernel. This makes it difficult to update or implement new protocols and technologies, since kernel programming requires bounded functions and operations. Deploying new emerging networking technologies would require kernel updates, which is not always a simple task and requires considerable time for debugging and adoption. In addition, the stack is implemented on kernel-POSIX-compliant-space, which brings inefficiencies such as system calls overhead and the use of a shared file descriptor space [6].

The Linux kernel has become not only another performance bottleneck for web applications, for example, but also an environment where it is hard to create new protocols that can be quickly deployed across the Internet and data centers. With this in mind, another category of solutions came into play. Firstly introduced in [72], it moves TCP/IP protocols processing from kernel to user space but without the complexity needed when implementing such features inside the kernel. Note that the term "TCP/IP protocols" used in the context of this

[1] https://www.kernel.org/doc/html/v4.18/networking/af_xdp.html

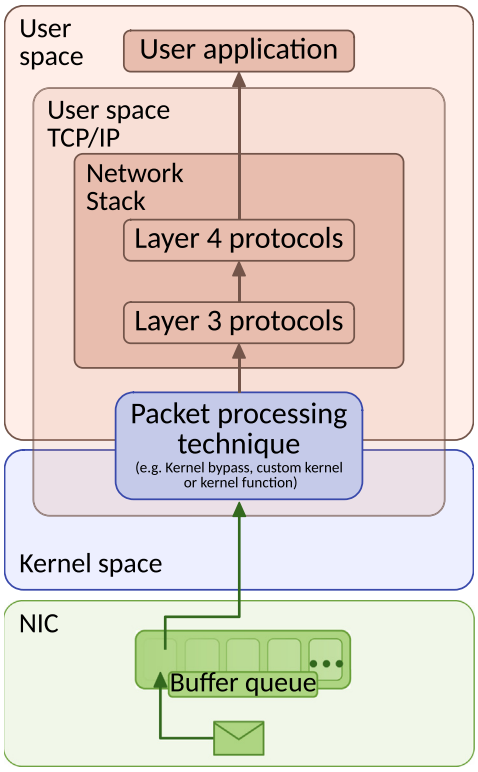

Fig. 9. Structure of User space TCP/IP solutions inside the Host.

study may refer to every protocol from layer three and above, or just individual protocols from the stack, when focusing on special-purpose applications. Some user space TCP/IP approaches combine technologies previously discussed, such as kernel bypass and zero-copy. In contrast, others create new alternative ways to process TCP/IP protocols in user space. Fig. 9 illustrates the general idea of user space TCP/IP structure.

5.4.1. mTCP

mTCP is a user-level TCP stack designed for multicore systems. It implements only the TCP protocol in user space [6], and designs a series of processes that interact with the protocol implementation, like sockets, buffers, and event queues. mTCP runs as a unique thread at the user level and provides library functions that allow applications to communicate with the thread. The framework offers backward compatibility, meaning that adapting existing applications to work with it requires minor modification from some kernel system calls to the mTCP API counterpart functions.

The mTCP design draws from three main optimization features: (i) it translates multiple expensive system calls into a single shared memory reference that the different cores may gain access to, (ii) allows efficient flow-level event aggregation, and (iii) performs batch packet processing for high I/O efficiency. Similar to DPDK, mTCP supports batch packet processing, which benefits memory usage by reading a batch of packets from a NIC. It also offers batching event handling (“user space System Call”), processing the packet batch in a batch of flow-level events.

Using a vanilla kernel to process packets from/to a NIC would cause performance degradation. To avoid this, mTCP often sits on top of software packets processing solutions like DPDK or netmap. Initially, mTCP used to work with a patched version of PacketShader [19], a software router implemented on user space that interacts with the network card. With time, it received support for other frameworks as well.

5.4.2. QUIC

Initially developed by Google in 2012, the Quick UDP Internet Connections (QUIC) protocol [73] sits on UDP while removing some of the traditional problems of TCP such as head of line blocking (HOL) and a lack of stream multiplexing. QUIC usage of UDP allows QUIC packets to pass through existing middleboxes and firewalls, which see QUIC packets as mere UDP packets. While using TCP congestion control mechanisms, QUIC allows the concurrent and reliable transmission of different objects in parallel streams. QUIC also provides for each stream to have its congestion window. As a result, a packet loss in one of these streams does not affect all of them and hence removes the classical packet head of line blocking suffered by TCP. HTTPv3 uses QUIC for the concurrent transport of web objects. In addition to these prerogatives, we can also highlight:

(i) Multiplexing at the application level allows the firing of multiple requests simultaneously in the same connection. Although both Google’s SDPY and HTTP/2 protocols multiplex the transfer of web objects, their transfer is serialized when using TCP at the transport layer. QUIC, on the other hand, maintains HTTP/2 multiplexing and transfers each web object in the form of an independent stream with its congestion control window. As a result, the HOL problem is mitigated. (ii) Decrease in the amount of “handshakes” necessary to establish a connection, which may be 0 RTTs if a cryptographic cookie is used, pre-stored in cache on the client (QUIC-Crypto). Note that using 0-RTT connection establishment based on previously stored credentials may not be secure and should be used with care. (iii) Security Native implementation of the TLS 1.3 protocol. This protocol spends a single RTT to establish a secure session, unlike TLS 1.2, which requires two RTTs and is less secure. (iv) QUIC portability does not depend on native operating system implementation. This is a distinct advantage of a user-level communication protocol implementation, as it uses specific kernel functions. (v) Flexibility as performing improvements, changes, testing, and deployment of this user space protocol is more straightforward. (vi) Justice: QUIC is fairer in the distribution of network resources [74]. In this way, network resources (like bandwidth) are shared more equally using QUIC. (vii) Traditional TCP congestion control mechanisms, such as Early Retransmit or Fast Retransmit, have been optimized for use in QUIC.

Due to its support by major browsers, QUIC has rapidly gained popularity. Some leading ISPs reported as much as reported 20% of their packets used the QUIC transport protocol. As of February 2021, around 5% of websites use HTTP/3, which relies on QUIC. But despite its broad industry support, QUIC suffers some limitations. In the context of packet error recovery, QUIC uses the Sliding Window Random Linear Code (RLC) and Forward Erasure Correction (FEC) techniques. Different QUIC deployments report disabling FEC due to the observed overhead that often offsets the gains.

Several authors have analyzed the performance of the QUIC protocol [74–78]. The study in [74] shows that higher QUIC throughput, when compared to TCP, does not necessarily translate into actual faster transfers. This is due to the high overhead of QUIC, possibly due to the user-level processing of packets. Furthermore, in ideal network scenarios with no packet loss or small delay, TCP outperforms QUIC.

5.4.3. Sandstorm

The Sandstorm [79] solution takes a different approach in the user space TCP/IP class of approaches. It proposes a *specialized network stack*, implementing a user space TCP/IP solution specialized in serving static web content, hence creating a *specialized* network stack for web servers. This same work also presents Namestorm, which takes a similar approach to the one from Sandstorm but specializes in the processing of packets of a different application, namely, a high-performance DNS service.

Sandstorm implements its specialized TCP/IP stack in user space, on top of the netmap framework introduced in 5. It leverages netmap as the traditional network packet I/O and event-notification capability

needed to process packets directly to/from NIC. Then it implements Ethernet, IP, TCP, and UDP protocols in user space with a focus on a static web service.

Sandstorm loads the static file from the web service into memory and generates all the packets that it will send, from the link to the TCP layers. Next, a "TCP-protocol control block (TCB)" manages every connection's state, allocating IP and TCP headers in a hash table to locate each TCP stream with the request through the TCB. Once located, Sandstorm increments the previously created packet with the information of the current request and sends the response directly to the NIC via netmap.

5.4.4. IX

The IX [80,81] team developed a different approach for user space TCP support. IX is an entirely new operating system based on data and control plane separation that uses the Linux Kernel. Here, the control functionality from the kernel, such as hardware management or resource configuration, separates from the data plane, such as network stack and application logic. The separation and isolation of these two planes, kernel (control plane) from network processing (data plane), is achieved by hardware virtualization. In the IX design, the kernel performs the control plane, responsible for provisioning, scheduling, and other control-related tasks.

In contrast, the data plane is transported to user space to run as a protected, library-based OS. The IX data plane operating system is optimized for high I/O performance. However, its control plane still benefits from the security and protection extended by the kernel.

IX processes packets with a zero-copy API that supports system calls and packet batching. Packets are only processed in a bounded batches mode during congestion, and the batch size is based on load. Applications may use the API to perform I/O and network operations. To optimize for both bandwidth and latency, IX dedicates hardware threads and networking queues to data plane instances. These threads must process packet batches to completion. Performance tests report that IX outperforms Linux and existing user space network stacks in terms of the achieved throughput and end-to-end latency.

6. Virtualization solutions

System *softwarization* is becoming the norm for the design of cloud systems, networks, data centers, and wireless technologies. The result is more flexibility and fine-grained control. We see this tendency in the design of Software-Defined Networks (SDN), the virtualization of network functions (NFV), the definition of Software-Based Data Centers, and last but not least, the development of Software-Defined Radio (SDR). Despite offering benefits like flexibility, elasticity, and ease of service and resource allocation, virtual environments often impose network processing degradation [82,83].

There have been numerous attempts made to counter this degradation issue. In this section, we examine and discuss some of them. We identified two main technology categories that serve virtual environments: **Physical Offloading** and **User Space Processing**.

6.1. Physical Offloading

This first category consists of solutions that attempt to offload some part or the entirety of the hypervisor network processing to other components of the system, like either the network card or the kernel. In particular, we look at these popular technologies: **PCI Passthrough** and **SR-IOV**.

6.1.1. PCI Passthrough

Through Intel's VT-d extension [84] (or IOMMU for AMD [85]), it is possible to introduce PCI devices from the native host system to the virtual guest OS. PCI Passthrough enables PCI devices to be directly assigned to VMs by the host. This way, a VM is connected to the PCI device as if it were physically connected to it [86]. This avoids the overheads of constant context switch from VM to host ("VM exits") and memory copies necessary by hypervisors to forward network packets from the NIC to the VMs, for example.

The way PCI Passthrough works in the Linux Kernel Virtual Machine (KVM) [87] is by emulating a device in the VM that is the equivalent to the real device. KVM then maps the address space of the physical device into an emulated address space in the VM used by the emulated device, making all the VM I/O access go directly to the device's hardware.

6.1.2. SR-IOV

The SR-IOV standard [88], published by the special interest group working on PCI (PCI-SIG), allows one PCI/PCIe device to present itself as multiple PCI/PCIe devices to the host. This is possible by slicing the physical device into multiple virtual ones, where each is presented to the PCI bus as a unique physical one. In this context, Virtual PCI Functions (VFs) refer to virtual devices, and the real physical ones are the Physical PCI Functions (PF). The number of virtual instances that can be used depends on the network adapter card and device driver.

Every VF has its own configuration space, with base address registers and memory space. It also has sent/receive queues, allowing each function to process its Direct Memory Access (DMA) and interrupts. These VFs are usually controlled by the kernel drivers. This means that manufacturers producing SR-IOV-enabled devices can implement virtual functions like existing physical devices to reuse existing drivers. This avoids the development of new drivers and helps the adaptations needed to adopt them. SR-IOV dedicates resources, such as processing cores, for each VF hence increasing its performance. This way, containers, and VMs can connect to a VF and retain a "dedicated NIC" to exchange packets.

The most significant advantage of SR-IOV is its association with PCI Passthrough. This combination directly assigns each VF to a single VM through its PCI address. Just like in *normal* PCI Passthrough, KVM maps the address registers of each PCI device to the address space of the VM, but with SR-IOV, it maps the address registers of the VFs. This way, multiple VMs can directly access the PCI device through the VFs, preventing overheads caused by the hypervisor, as discussed earlier. Fig. 10 shows how SR-IOV + PCI Passthrough works in comparison with vanilla virtual network drivers.

It is possible to find some works that evaluate the use of SR-IOV in packet processing scenarios [89,90], where they generally conclude that SR-IOV can yield better performance especially to full virtualization VMs, and even more so when combining SR-IOV with PCI Passthrough.

6.2. User space processing

This category consists of solutions that create a user space environment to process and deliver packets directly to VMs. In this category, we survey vHost-User and FD.io VPP.

6.2.1. VHost-user

When considering communication in virtual environments, both the VM and the native host must have driver implementations that enable and integrate their communication in order to improve performance. In a scenario lacking acceleration resources, such as SR-IOV or PCI Passthrough, one generally uses the VirtIO para-virtual driver [91].

VirtIO inserts and allows VMs to use mechanisms such as "Virtqueues" and "virtio ring". The last one of interest to this work is an exciting feature that helps transmit/receive queues to be shared with the hypervisor. Thus, when some packets are received in these queues,

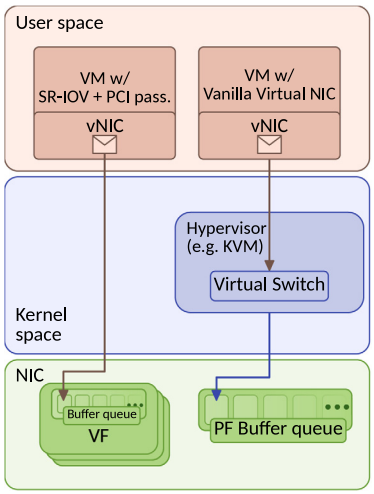

Fig. 10. VMs using SR-IOV + PCI Passthrough vNICs vs. Vanilla vNICs.

the hypervisor forwards them to the native host's network. However, the communication between the guest VM and the hypervisor occurs in the user space, causing the need for more data copies and more frequent CPU privilege context exchanges, especially if we consider a scenario where we have packets of small size. This feature ends up limiting performance and creating a bottleneck.

In this scenario, VHost was developed to overcome this problem caused by the need to fully use the hypervisor as a communication intermediary between the guest VM and the native host's network. Thus, VHost allows exchanging data between the native host kernel and the guest VM more directly, sharing the virtqueue without the hypervisor taking over the packet processing. This optimizes network I/O performance. It is essential to highlight that the hypervisor will still maintain some of its role in this communication process, mainly of a control-level nature, such as configuring the virtual queues on the NICs.

VHost-user has multiple similarities to VHost. It differentiates itself from VHost mainly in terms of the implementation placement, as VHost works in the Linux kernel, as opposed to the user space in the case of VHost-user. The need to apply the mechanisms offered by VHost in the user space was essentially due to the demand for scenarios where there is communication between processes at the user level, such as technologies that are embedded or converging with the SDN or NFV paradigm. For example, if we employ VHost in these scenarios, there would be a need for data copies between the guest and host since VHost operates at the kernel level, creating a new bottleneck. VHost-user overcomes this limitation.

6.2.2. FD.io VPP

FD.io VPP (Vector Packet Processing) [92,93] focuses on advancing cloud computing technology, taking an approach aimed especially at the SDN/NFV field. VPP creates an environment to enable fast packet processing and delivery to user space applications, more precisely network functions, in the form of VNFs or CNFs. Although VPP aims to be application agnostic, meaning it can serve bare-metal or virtual applications, it has gained attention in virtual functions deployment, moving its development towards a cloud-centered environment and creating native support for platforms like OpenStack or Kubernetes. FD.io VPP serves different purposes, being flexible to attend to other use cases like traffic management, stateful and stateless load-balancing, or intrusion prevention [94]. Also, multiple network functions are already available, including DHCP, MPLS, VLAN, ARP, IPv6, etc. In addition, implementing new functions is also available in the form of "plug-in nodes", using VPP's API [95].

FD.io VPP mainly runs on top of DPDK as the official "driver", although it also supports netmap. VPP refers to DPDK as the NIC driver, as it is responsible for interacting with the network card to send and receive packets at high speed in user space [96]. Since it uses DPDK, VPP can run on user space without relying on kernel interaction. This enables its utilization on commodity CPU, which eases its compatibility across data centers.

VPP takes advantage of the batch packet processing from DPDK and yields the Vector Processing across all its functions. Instead of processing one packet throughout the whole network stack, VPP takes a vector with multiple packets and processes them through all available functions in its network stack. The vector size is flexible, changing according to the network load: if there is one packet to process, the vector size is one packet long. If the load increases and ten packets arrive, the vector size increases to 10 until it reaches the maximum number of packets the system can process. This approach decreases the processing cost per packet and amortizes the CPU cache processing [97].

As mentioned, VPP processes packets through the available functions in its network stack. The "available functions" relate to the fact that VPP uses a Packet Processing Graph. Each function is treated as a node, usually related to a stack protocol, put in a graph architecture that determines the available network stack functions that which the packets will be processed. So, at runtime, the processing graph is applied to a newly created vector of packets, and the vector goes through the graph nodes until reaching the user application.

7. Discussion

This section discusses implementations of the packet processing approaches introduced in this article, analyzing how they behave in real-world scenarios. To achieve this, we discuss each class of solution under the following four different criteria: (i) host resource usage, (ii) high packet rate, (iii) system security and (iv) flexibility/expandability. We choose these four criteria according to the design requirements addressed the most when developing and/or testing a packet processor solution, as seen in works like [6,7,35,63,80,98]. Afterwards, we also discuss some real-world use-cases that combine different solutions explained in this survey.

Table 2 maps each literature solution to the criterion they satisfy. It is possible to see that some technologies cannot fulfill an entire criterion, which may indicate that the technology simply does not comply with said criterion, or that the authors could not find studies that indicated a solution that could satisfy such criterion.

7.1. Host resource usage efficiency

Under this criterion, we discuss how efficiently each solution uses the host's resources. Hardware-based approaches show some efficiency as they mainly offload packet processing out of the host and into the NIC, as in TOEs and Programmable NICs. This enables two main benefits. First, the host CPU is not overloaded by network processing and can dedicate its processing power to other tasks. Second, as the NIC is dedicated to performing specific tasks, it may outperform the host [99]. As seen in [100], almost 125 CPU cycles can be reduced when using hardware offloading functions. The work in [101] creates a programmable NIC to offload Key-Value store access from the host. The experiments show that when the programmable NIC is at peak load, there is a minimal impact on other workloads on the server. In addition, [11] combines programmable NICs and RSS to achieve a high throughput scalable system that also prioritizes CPU usage. They program the NIC to allocate cores and schedule packet processing, freeing the CPU cores from scheduling tasks while enabling batch jobs and processing other tasks, and also achieving a CPU load that scales according to the incoming packet rate. Also, offloading tasks to programmable NICs can decrease the data exchange between NIC and the host system, which reduces the usage of PCIe bus that can impose a significant performance cost to the system, as seen in [102]. Alongside this, APUNet [49] stands out as an offloading solution that integrates

Table 2
Summary of packet processing literature discussed with their respective criterion.

Architecture	Criterion	Literature
Hardware	Host Resource Usage	[49] [99] [100] [101] [11] [102]
	High Packet Rate	[103] [104] [50] [105] [98] [9] [106] [36]
	System Security	–
	Flexibility/Expandability	–
Software	Host Resource Usage	[107] [7] [63] [67] [108] [26]
	High Packet Rate	[109] [63] [110]
	System Security	[63] [73] [7] [80]
	Flexibility/Expandability	[63] [73]
Virtualization	Host Resource Usage	[86]
	High Packet Rate	[111] [92] [93]
	System Security	[112] [86]
	Flexibility/Expandability	[113]

CPUs with GPUs in the same chip, removing a bottleneck in the PCIe used for communication between a CPU and the GPU dedicated for packet processing.

The zero-copy mechanism offers reasonable resource usage by freeing CPU cycles regarding software-based solutions. The proposal presented in [26] shows that, with a buffer size of 4KiB, zero-copy can process a TCP stream with 79% of the system cycles that a normal equivalent copy process would use. If the buffer size is more prominent, as for 1MiB, this value falls down to 61%. Thus, although it benefits standard kernel processing, zero-copy is not sufficient on its own to guarantee ideal resource efficiency. Also, as said previously, MSG_ZEROCOPY only supports zero-copy on transmitting packets, which is not ideal for most data center environments. This is why zero-copy is mainly used jointly with other mechanisms, as shown in Section 5.

The Kernel Bypass approach usually uses techniques that reduce resource consumption like batch packet processing [107], often referred to as *bulk* or *burst* packet processing. Suppose a packet processor can only process one packet at a time. In this case, the host resource usage will not be as efficient since all CPU cycles, interrupts, and memory allocation used to process that packet are dedicated to only a single packet. The batch processing approach processes more than one packet per API call. This reduces memory usage, API calls, and PCIe and I/O bus transactions. DPDK and netmap adopt this technique as their base functionality.

Another technique used as part of the kernel bypass is the elimination of System Calls. This is beneficial since System Calls introduce a processing overhead due to frequent mode and context switching and processor state pollution, as discussed in Section 2. DPDK and PF_RING implement this feature. However, netmap still uses some System Calls from the kernel to perform I/O synchronization or memory access. Additionally, one may combine both kernel bypass and the removal of system calls with a zero-copy mechanism inside the bypass path to avoid memory consumption.

Although the kernel bypass solutions use the aforementioned techniques, these solutions are not always the most resource-efficient. Most of them usually pin CPU cores to the application in user space [7,63] and, after all, maintain the processing of the packets inside the host. The main goal of bypass solutions is not to be resource-efficient. It is an efficient packet processor, as opposed to XDP, that aims at a trade-off of resource usage efficiency and high throughput performance. This trade-off is depicted in [63], where it is possible to see that DPDK outperforms XDP in throughput performance but requires heavier CPU usage due to its busy-polling driver. In exchange, XDP achieves lower throughput performance but uses less CPU, scaling CPU usage with the incoming packet rate. However, this behavior is not mandatory, as shown in [67]. The evaluation of two Memcached implementations using DPDK and XDP shows that the XDP implementation can achieve similar throughput for Memcached requests while maintaining low CPU usage and scaling the CPU usage according to the request rate. The work in [108] brings a model for packet processing by applying a thread sleep in DPDK to enable a lower CPU usage in cluster environments. The results show a reduction of approximately 80% of CPU usage in some cases.

7.2. High packet rate

The high packet rate criterion discusses the surveyed solutions based on the highest achievable packet processing rate. The hardware architecture category does not consistently achieve high performance. TOEs can achieve a rate close to 10 Gbps, or around 8.5 Gbps to 9.9 Gbps as reported in [36,106], respectively. As for Programmable NICs, the achievable throughput is variable. In [105], the experiments conducted show that a line-rate performance is possible. Still, it depends on the network applications running inside the NIC, so the number of match+action tables used in the pipeline or the number of cryptography operations can influence the packets per second throughput. Even though programmable NICs have physical processing capabilities (line-rate) of 40 Gbps or even 100 Gbps, their packet rate performance can be lowered if the offloaded tasks are resource hungry. As programmable NICs resources like CPU or RAM are limited, this limitation can increase the per-packet processing time. This means the offloaded task must be well tested. As the CPU inside the programmable NICs is slower than that of the host, specific heavier processing can lose performance if offloaded, like heavy-loaded CPU tasks such as encryption or checksum calculation, as seen in [9,98].

Despite the advantages of GPUs in packet processing, there are some limitations to the gains they provide. Performance gains may be offset by the fact that not all packet processing tasks can execute in parallel and hence cannot benefit from the usage of GPUs. In addition, some types of processing may not be suitable for execution by GPUs [103]. Examples include complex conditional processing and frequent branching. A programmer may then need to spend an additional effort in optimizing processing. Examples include the batch processing packets that follow the same execution path and perform lightweight processing of some bytes in a packet. Some limited levels of pre-processing by the CPU may help a GPU overcome these problems. For example, an application may selectively select packets that follow the same processing path and pass these to the GPU. There must be a balance between the benefits of such cooperation and the cost of transferring packets between the CPU and GPU. In summary, GPUs are not suitable for dealing with low flow rates. On the other hand, they outperform CPUs in intensive processing applications such as intrusion detection [104] and packet encryption [50].

The kernel bypass and XDP are the most noticeable frameworks for reaching high rates in the software architecture category. Most kernel bypass solutions can saturate a 10 Gbps link [109], and specially DPDK and XDP are known to saturate 40 Gbps, and 100 Gbps network interfaces [63,110]. It is important to indicate, however, that depending on the environment and workload, DPDK can perform higher packet processing rates than XDP [98,114].

Considering the design of virtualization architectures, their main goal is not to achieve a higher packet processing rate but to act as an enabling technology that supports high packet rates inside the VMs by eliminating any possible virtualization overhead [111]. This depends mainly on the physical throughput of the network card and the application running inside the VM.

7.3. System security

This criterion investigates how each solution preserves the system's integrity by performing well-isolated tasks and not overloading processing. Solutions from the hardware category raise significant concern, especially because they introduce new hardware-based implementations that deliver packets ready to be *directly* used by applications in user space. This is the case for TOEs. This means that implementing such devices should come with a well-defined device driver. As device drivers are the main connection point between the TOEs and the Operating System, they are responsible for the host memory access and Hard/Soft Interrupts. Also, device drivers require kernel manipulation, which must be done according to kernel standards, avoiding the need for "custom" kernel distributions. As for Programmable NICs, besides the security issue highlighted above, offloading tasks to the NICs should come with a protective design to verify these programs to ensure the protection of the host and its resources.

In the software architecture category, providing system security requires more elements to consider. The zero-copy approach is the simplest one under this criterion, since its implementation is inside the kernel. This provides all the kernel security and isolation necessary to maintain the system's safety. On the other hand, kernel bypass techniques are more critical. When excluding the kernel from the packet processing path, the default kernel's security is also consequently excluded. Thus, it is up to the framework to reassure isolated resource usage, bounded memory access, and finite program execution [63].

XDP takes similar considerations as zero-copy. Implemented inside the kernel, it runs by default with all security provided by the kernel. Furthermore, it uses the eBPF execution environment and verifies all programs before executing, creating another layer of security. It is worth highlighting that the eBPF verifier is a major security component in the kernel. As it is responsible for allowing what is secure or not to run inside the kernel, any bugs in the verifier that would allow unsafe code to execute in the kernel could present a great threat to the system. So the kernel must be continuously updated and verified to ensure such concern. The user space TCP technology takes the same considerations as the kernel bypass, since it uses bypass technologies like DPDK or netmap to work.

In the case of the used virtualization architectures, the solutions are not completely secure from security issues, as seen in [86,112]. The works show how implementations of PCI Passthrough are prone to DoS attacks. A Virtual Machine that launches a DoS attack on other local VMs can degrade every other I/O device that shares the PCI link with the DoS victim, including the VMs that uses these I/O devices.

7.4. Flexibility/expandability

Regarding the flexibility and expandability criterion, we examine the flexibility and ease to expand an environment of a given class of solutions. Understandably, the hardware class of solutions is the least flexible one, as in the case of TOEs. As they are hardware implementations with their processing programmed directly on silicon, installing, changing, or upgrading them requires the acquisition or replacement of hardware with the new changes installed. This is not always easy to perform, especially in a Data Center environment with numerous servers and NICs to change. Programmable NICs are more flexible up to a certain degree. As they are programmable, the administrator can reprogram and update the NIC's functionality logically. However, as Programmable NICs have limited resources if compared to the host, these updates are also limited. Not only this, specific changes are not possible to implement by software upgrades.

The software class of solutions is generally more flexible. The zero-copy technology is implemented entirely in software inside the kernel and is hardware-independent. Also, building applications that use the zero-copy module requires minor upgrades, limited to only changing specific System Calls. The Kernel Bypass technology and XDP are not hardware dependent, but "device driver dependent". To work and talk to the NIC, they depend on either a device driver's implementation in user space or a kernel space driver modification. If an organization does not own NICs compatible with the supported drivers, they must acquire compatible ones or implement a driver to their current owned NICs. In addition, as kernel bypass solutions take complete control of the network hardware, they make it harder to scale up with external applications not using the bypass solutions, isolating the internal application from the rest of the system [63].

In the virtualization category, the hardware passthrough suffers from the same issues as the TOEs. It implements its functionalities directly on the hardware, which means that scaling a PCI pass-through infrastructure, for example, depends directly on hardware swapping. Also, as studied in [113], VMs with PCI Passthrough are not able to perform the live migration of VMs by default, which represents a significant feature in cloud computing data centers that is lost.

We highlight that most comments on user space TCP and software passthrough are similar to the Kernel Bypass ones, mainly because both of these solutions are built on top of Bypass solutions.

7.5. Real-world solution combinations

The studied solutions in this survey have the main purpose of somehow enhancing packet processing in commodity hardware systems. The real advantage of such solutions is put to the test when used in real scenarios. Multiple times, these scenarios tend to combine different technologies to achieve the most optimal configuration for that case. We review some of these combinations next.

Open vSwitch (OVS) [115] is a software switch used as a virtual switch within virtual environments. OVS supports standard switch capabilities, but it stands out for supporting SDN capability, with support to protocols like OpenFlow and OVSDB [116]. A recent enhancement to OVS has been done towards adding native DPDK support to OVS. This configuration can further enhance OVS's performance by introducing a fast user space processing to deliver a packet to OVS. The work in [117] investigates the performance of OVS-DPDK with SR-IOV and FD.io. It shows that the performance of OVS-DPDK is close to that of FD.io in most cases, although SR-IOV can outperform both since it consists of physical configuration. As in [118,119], the authors perform a comparison between OVS and OVS-DPDK – among other virtual switches – and show a significant performance gain when using OVS-DPDK. For example, when processing small 64B packets, OVS can only process around 1.5 Mpps, while OVS-DPDK can achieve around 16 Mpps processing throughput. Also, in [120], OVS-DPDK is tested with flow and port mirroring, representing a common feature in network management. The results show OVS-DPDK maintaining low switching latency and a high flow throughput. With high incoming throughput, there is a higher deviation in latency.

The work in [121] presents VIRTIO-USER, a standardized channel for fast network communication for containers. It combines the vhost adapter with VirtIO standardization and DPDK fast packet processing to create a user space network communication for containers. It can overcome standard kernel base networking, achieving a throughput 7x higher than kernel base network when processing 64B packets.

Another combination is the possibility of using XDP with the Linux RSS feature. This combination can be seen in [63,98] and also [69]. They use the RSS feature to steer packets to multiple cores and assign the XDP program to run in each core, enabling it to process packets in parallel and thus further increase performance.

The work in [122] evaluates VNF performance when implemented using LibPCAP and DPDK connected with SR-IOV NIC. The results show that combining DPDK and SR-IOV can enhance the performance of the VNFs, enabling the network function to achieve close to line-rate throughput performance.

In [123] a similar work is done. The authors combine the previously discussed combination of Open vSwitch and DPDK (OVS-DPDK) with

the vhost-user protocol in a virtual KVM environment. OVS-DPDK acts as the virtual switch for every VM on the server with all DPDK features to enhance packet processing. The vhost-user protocol enhances packet exchange between VMs and OVS-DPDK, further improving packet processing. The evaluation performed showed a 13.02 Mpps forwarding throughput performance, close to the line rate of 14.88 Mpps.

The work shown in [124] displays the development of GRO and GSO user space libraries for DPDK. They implement a GRO Reassembly Algorithm responsible to merge packets and a GSO Segmentation Scheme that splits packets, being responsible for the merging and segmentation of packets inside DPDK. This can be ideal since these libraries rely only on software implementation and need no hardware feature to work, and since DPDK is a user space packet processing, it fits as a good solution. The experiments show that DPDK GRO can deliver 1.9x the performance of Linux GRO and DPDK-GSO 1.5x the performance of Linux GSO.

Another interesting point is P4 and XDP. P4 [125] is a declarative high-level language for programming protocol-independent network devices. P4 started mainly towards bringing programmability inside switches in the SDN field, creating a device and protocol-independent environment. A programmer should not need to know specifics of the target hardware the program would run on, and that the switches should not be tied to network protocols. Lately, P4 is making its way towards programmable NICs, bringing the same goals to network cards [126]. P4 and XDP share a common feature when analyzing the capability of XDP to export programs to programmable NICs [127]. In these cases, XDP can deliver a higher packet processing performance with half CPU usage, with the trade-off that P4 programs are more straightforward with less amount of code lines. Also, as P4 aims at being device and protocol independent, creating a specific application needs little or no modification when changing devices or scenarios. With this in mind, P4C-XDP was created, a P4 compiler that targets XDP programs [128,129]. It compiles P4 programs into stylized C programs that can, in turn, be compiled into eBPF programs, bringing the benefits of both tools.

8. Conclusions

This article provides an overview of the processing of network packets in Linux environments, describing the characteristics and how this mechanism works in detail, while also depicting the flaws and disadvantages of this processing. We then present the attempts to solve these problems, which vary in terms of system architecture and technologies used. We categorize these solutions and explain how they work, then discuss their natural world use-case peculiarities and usage.

Though these solutions have been described and compared, there is no silver bullet. Datacenter managers, network operators, system administrators, and application developers must carefully weigh the pros and cons of each of the approaches according to their scenario. Hardware solutions may be helpful to lower host resource usage, but may also be challenging to scale up one's infrastructure. Software solutions like kernel bypass or User Space TCP can avoid kernel stack overhead and provide a fast packet processing environment, but introduce constraints on system security and OS isolation and interaction. A Hardware solution for virtualization can also suffer from the flexibility and expandability point of view. Still, it can provide direct access to the network card from a VM, whereas software solutions for virtualization suffer from OS isolation and interaction problem. Nonetheless, they may also achieve fast packet processing rates inside VMs. As stated, network operators and system administrators should understand how to combine some of these solutions to reach a setup that responds to the needs of their specific usage scenario.

This paper is the first to provide a review that puts together the recent advances in packet processing to the best of our knowledge. As such, we hope the reader finds it helpful to demystify some of the concepts and point to the road ahead.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgments

This work was supported by Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), Brazil, Fundação de Amparo à Ciência e Tecnologia de Pernambuco (FACEPE), Brazil and Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), Brazil.

References

[1] B.A.A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, T. Turletti, A survey of software-defined networking: Past, present, and future of programmable networks, IEEE Commun. Surv. Tutor. 16 (3) (2014) 1617–1634, http://dx.doi.org/10.1109/SURV.2014.012214.00180.

[2] D.. Reuse, Cadence expands proven ethernet IP offering with 40/100 gigabit ethernet solution, 2012, Website URL https://www.design-reuse.com/news/28520/40-100-gigabit-ethernet-mac-pcs-ip-core.html.

[3] J. D'Ambrosia, D. Law, M. Nowell, 40 Gigabit ethernet and 100 gigabit ethernet, 2010, p. 13, Website URL https://www.ethernetalliance.org/wp-content/uploads/2011/10/document_files_40G_100G_Tech_overview.pdf.

[4] C. Public, The future is 40 gigabit ethernet, 2016, p. 7, On-line White Paper URL https://www.cisco.com/c/dam/en/us/products/collateral/switches/catalyst-6500-series-switches/white-paper-c11-737238.pdf.

[5] L. Foundation, Data plane development kit, 2018, URL https://www.dpdk.org/.

[6] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, K. Park, mTCP: a highly scalable user-level TCP stack for multicore systems, in: 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 14, USENIX Association, Seattle, WA, 2014, pp. 489–502.

[7] L. Rizzo, Netmap: A novel framework for fast packet I/O, in: 2012 USENIX Annual Technical Conference, USENIX ATC 12, USENIX Association, Boston, MA, 2012, pp. 101–112.

[8] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, K. Gupta, Offloading distributed applications onto SmartNICs using IPipe, in: Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 318–333, http://dx.doi.org/10.1145/3341302.3342079.

[9] S. Miano, R. Doriguzzi-Corin, F. Risso, D. Siracusa, R. Sommese, Introducing SmartNICs in server-based data plane processing: The DDoS mitigation use case, IEEE Access 7 (2019) 107161–107170, http://dx.doi.org/10.1109/ACCESS.2019.2933491.

[10] P. Enberg, A. Rao, S. Tarkoma, Partition-aware packet steering using XDP and EBPF for improving application-level parallelism, in: Proceedings of the 1st ACM CoNEXT Workshop on Emerging in-Network Computing Paradigms, ENCP '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 27–33, http://dx.doi.org/10.1145/3359993.3366766.

[11] A. Rucker, M. Shahbaz, T. Swamy, K. Olukotun, Elastic RSS: Co-scheduling packets and cores using programmable NICs, in: Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019, APNet '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 71–77, http://dx.doi.org/10.1145/3343180.3343184.

[12] T.L.K. Organization, The linux kernel documentation, 2022, https://docs.kernel.org/ (Accessed 15 jan 2022).

[13] W. Wu, M. Crawford, M. Bowden, The performance analysis of linux networking – packet receiving, Comput. Commun. 30 (5) (2007) 1044–1057, http://dx.doi.org/10.1016/j.comcom.2006.11.001, Advances in Computer Communications Networks.

[14] A. Rubini, J. Corbet, Mmap and DMA, from Linux Device Drivers 2nd Edition, O'Reilly Media, Inc., 2001, https://www.xml.com/ldd/chapter/book/ch13.html.

[15] P. Emmerich, D. Raumer, A. Beifuß, L. Erlacher, F. Wohlfart, T.M. Runge, S. Gallenmüller, G. Carle, Optimizing latency and CPU load in packet processing systems, in: 2015 International Symposium on Performance Evaluation of Computer and Telecommunication Systems, SPECTS, 2015, pp. 1–8, http://dx.doi.org/10.1109/SPECTS.2015.7285275.

[16] R. Chen, G. Sun, A survey of kernel-bypass techniques in network stack, in: Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence, CSAI '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 474–477, http://dx.doi.org/10.1145/3297156.3297242.

[17] G.R. Wright, W.R. Stevens, TCP/IP Illustrated, Volume 2 (Paperback): The Implementation, Addison-Wesley Professional, 1995.

[18] P. Research, Linux networking stack from the ground up, 2016, URL https://www.privateinternetaccess.com/blog/linux-networking-stack-from-the-ground-up-part-1/.

[19] S. Han, K. Jang, K. Park, S. Moon, Packetshader: A GPU-accelerated software router, in: Proceedings of the ACM SIGCOMM 2010 Conference, SIGCOMM '10, Association for Computing Machinery, New York, NY, USA, 2010, pp. 195–206.

[20] J. Li, N.K. Sharma, D.R.K. Ports, S.D. Gribble, Tales of the tail: Hardware, OS, and application-level sources of tail latency, in: Proceedings of the ACM Symposium on Cloud Computing, SOCC '14, Association for Computing Machinery, New York, NY, USA, 2014, pp. 1–14, http://dx.doi.org/10.1145/2670979.2670988.

[21] Tsuna, How long does it take to make a context switch?, 2010, Internet URL https://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html (Accessed: January, 2021).

[22] C. Li, C. Ding, K. Shen, Quantifying the cost of context switch, in: Proceedings of the 2007 Workshop on Experimental Computer Science, ExpCS '07, Association for Computing Machinery, New York, NY, USA, 2007, pp. 2–es, http://dx.doi.org/10.1145/1281700.1281702.

[23] L. Soares, M. Stumm, Flexsc: Flexible system call scheduling with exception-less system calls, in: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI '10, USENIX Association, 2010.

[24] T. Hruby, T. Crivat, H. Bos, A.S. Tanenbaum, On sockets and system calls: Minimizing context switches for the socket API, in: 2014 Conference on Timely Results in Operating Systems, TRIOS 14, USENIX Association, Broomfield, CO, 2014, URL https://www.usenix.org/conference/trios14/technical-sessions/presentation/hruby.

[25] L. Tianhua, Z. Hongfeng, C. Guiran, Z. Chuansheng, The design and implementation of zero-copy for linux, in: 2008 Eighth International Conference on Intelligent Systems Design and Applications, Vol. 1, 2008, pp. 121–126, http://dx.doi.org/10.1109/ISDA.2008.102.

[26] W.D. Bruijn, E. Dumazet, Sendmsg copy avoidance with msg_Zerocopy, in: NetDev, the Technical Conference on Linux Networking, in: Netdev 2.1, Netdev, Montreal, Canada, 2017, URL https://netdevconf.info/2.1/papers/debruijn-msgzerocopy-talk.pdf.

[27] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, G. Carle, Comparison of frameworks for high-performance packet IO, in: Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS '15, IEEE Computer Society, USA, 2015, pp. 29–38, http://dx.doi.org/10.1109/ANCS.2015.7110118.

[28] A. Currid, TCP offload to the rescue, Queue 2 (3) (2004) 58–65, http://dx.doi.org/10.1145/1005062.1005069.

[29] J. Corbet, Batch processing of network packets, 2018, URL https://lwn.net/Articles/763056/.

[30] T.L.K. Organization, Linux (version 2.5.7), 2022, https://cdn.kernel.org/pub/linux/kernel/v2.5/ChangeLog-2.5.7 (accessed 15 jan 2022).

[31] A.T. de Oliveira Filho, E. Freitas, P.R.X. Carmo, D.H. Sadok, J. Kelner, An experimental investigation of round-trip time and virtualization, Comput. Commun. (2021) http://dx.doi.org/10.1016/j.comcom.2021.12.006, URL https://www.sciencedirect.com/science/article/pii/S0140366421004722.

[32] T. Herbert, W. de Bruijn, Scaling in the linux networking stack, 2019, Internet URL https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[33] I. Corporation, Improving Network Performance in Multi-Core Systems, Tech. rep., Intel Corporation, 2007, URL https://www.intel.com/content/dam/support/us/en/documents/network/sb/318483001us2.pdf.

[34] T. Herbet, Rps: Receive packet steering, 2010, Mailing List URL https://lwn.net/Articles/370153/.

[35] J.C. Mogul, TCP offload is a dumb idea whose time has come, in: Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9, HOTOS '03, USENIX Association, USA, 2003, p. 5.

[36] L. Ding, P. Kang, W. Yin, L. Wang, Hardware TCP offload engine based on 10-gbps ethernet for low-latency network communication, in: 2016 International Conference on Field-Programmable Technology, FPT, 2016, pp. 269–272, http://dx.doi.org/10.1109/FPT.2016.7929550.

[37] T. Uchida, Hardware-based TCP processor for Gigabit ethernet, IEEE Trans. Nucl. Sci. 55 (3) (2008) 1631–1637, http://dx.doi.org/10.1109/TNS.2008.920264.

[38] Y. Moon, S. Lee, M.A. Jamshed, K. Park, AccelTCP: Accelerating network applications with stateful TCP offloading, in: 17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 20, USENIX Association, Santa Clara, CA, 2020, pp. 77–92, URL https://www.usenix.org/conference/nsdi20/presentation/moon.

[39] R. Shashidhara, T. Stamler, A. Kaufmann, S. Peter, Flextoe: Flexible TCP offload with fine-grained parallelism, 2021, arXiv preprint arXiv:2110.10919.

[40] Intel Corporation, Intel 82576eb gigabit ethernet controller datasheet, 2011, Rev. 2.63.

[41] K. Team, Segmentation Offloads in the Linux Networking Stack, Internet URL https://www.kernel.org/doc/Documentation/networking/segmentation-offloads.txt.

[42] Y. Cheng, N. Cardwell, Making linux TCP fast, in: Netdev Conference, 2016.

[43] W. de Bruijn, E. Dumazet, Optimizing UDP for content delivery: GSO, pacing and zerocopy, in: Linux Plumbers Conference, 2018.

[44] J. Pinkerton, Sockets direct protocol V1.0, 2003.

[45] Rsocket(7) - linux man page, 2019, https://linux.die.net/man/7/ (Acessed 15 jan 2022).

[46] B. Li, T. Cui, Z. Wang, W. Bai, L. Zhang, Socksdirect: Datacenter sockets can be fast and compatible, in: Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 90–103, http://dx.doi.org/10.1145/3341302.3342071, URL.

[47] Y.G. Moon, I. Park, S. Lee, K.S. Park, Accelerating flow processing middleboxes with programmable NICs, in: Proceedings of the 9th Asia-Pacific Workshop on Systems, APSys '18, Association for Computing Machinery, New York, NY, USA, 2018, http://dx.doi.org/10.1145/3265723.3265744.

[48] G.P. Katsikas, T. Barbette, M. Chiesa, D. Kostić, G.Q. Maguire, What you need to know about (smart) network interface cards, in: International Conference on Passive and Active Network Measurement, Springer, 2021, pp. 319–336.

[49] Y. Go, M. Jamshed, Y. Moon, C. Hwang, K. Park, Apunet: Revitalizing GPU as packet processing accelerator, in: Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation, NSDI '17, USENIX Association, USA, 2017, pp. 83–96.

[50] G. Vasiliadis, L. Koromilas, M. Polychronakis, S. Ioannidis, GASPP: A GPU-accelerated stateful packet processing framework, in: Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference, in: USENIX ATC'14, USENIX Association, USA, 2014, pp. 321–332.

[51] C. Jung, S. Kim, I. Yeom, H. Woo, Y. Kim, GPU-ether: GPU-native packet I/O for GPU applications on commodity ethernet, in: IEEE INFOCOM 2021 - IEEE Conference on Computer Communications, 2021, pp. 1–10, http://dx.doi.org/10.1109/INFOCOM42981.2021.9488699.

[52] M. Daga, A.M. Aji, W.-c. Feng, On the efficacy of a fused cpu+gpu processor (or APU) for parallel computing, in: 2011 Symposium on Application Accelerators in High-Performance Computing, 2011, pp. 141–149, http://dx.doi.org/10.1109/SAAHPC.2011.29.

[53] K. Team, PCI Peer-to-Peer DMA Support, Internet URL https://www.kernel.org/doc/html/v5.0/_sources/driver-api/pci/p2pdma.rst.txt.

[54] J. Corbet, Zero-copy networking, 2017, URL https://lwn.net/Articles/726917/.

[55] H. Redžović, M. Vesović, A. Smiljanić, M. Bjelica, Energy-efficient network processing based on netmap framework, Electron. Lett. 53 (6) (2017) 407–409, http://dx.doi.org/10.1049/el.2016.3815.

[56] L. Rizzo, M. Carbone, G. Catalli, Transparent acceleration of software packet forwarding using netmap, in: 2012 Proceedings IEEE, INFOCOM, IEEE, 2012, http://dx.doi.org/10.1109/infcom.2012.6195638.

[57] V. Maffione, L. Rizzo, G. Lettieri, Flexible virtual machine networking using netmap passthrough, in: 2016 IEEE International Symposium on Local and Metropolitan Area Networks, LANMAN, IEEE, 2016, http://dx.doi.org/10.1109/lanman.2016.7548852, URL.

[58] L. Rizzo, G. Lettieri, VALE, a switched ethernet for virtual machines, in: Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies, CoNEXT 12, ACM Press, 2012, http://dx.doi.org/10.1145/2413176.2413185.

[59] Q. Ren, L. Zhou, Z. Xu, Y. Zhang, L. Zhang, PacketUsher: Exploiting DPDK to accelerate compute-intensive packet processing, Comput. Commun. 161 (2020) 324–333, http://dx.doi.org/10.1016/j.comcom.2020.07.040, URL https://www.sciencedirect.com/science/article/pii/S0140366420318521.

[60] D. Vladislavic, D. Huljenic, J. Ozegovic, Enhancing vnf's performance using DPDK driven OVS user-space forwarding, in: 2017 25th International Conference on Software, Telecommunications and Computer Networks, SoftCOM, 2017, pp. 1–5, http://dx.doi.org/10.23919/SOFTCOM.2017.8115534.

[61] R. Kawashima, H. Nakayama, T. Hayashi, H. Matsuo, Evaluation of forwarding efficiency in NFV-nodes toward predictable service chain performance, IEEE Trans. Netw. Serv. Manag. 14 (4) (2017) 920–933, http://dx.doi.org/10.1109/TNSM.2017.2734560.

[62] DPDK, DPDK docs: Programmer's guide, overview, 2017, URL http://doc.dpdk.org/guides/prog_guide/overview.html.

[63] T. Høiland-Jørgensen, J.D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, D. Miller, The express data path: Fast programmable packet processing in the operating system kernel, in: Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies, CoNEXT '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 54–66, http://dx.doi.org/10.1145/3281411.3281443.

[64] ntop, Pf_Ring ZC (zero copy) guide, 2018, URL https://www.ntop.org/guides/pf_ring/zc.html.

[65] M.A.M. Vieira, M.S. Castanho, R.D.G. Pacífico, E.R.S. Santos, E.P.M.C. Júnior, L.F.M. Vieira, Fast packet processing with EBPF and XDP: Concepts, code, challenges, and applications, ACM Comput. Surv. 53 (1) (2020) http://dx.doi.org/10.1145/3371038.

[66] C. Authors, BPF and XDP reference guide, 2017, URL https://docs.cilium.io/en/latest/bpf/#bpf-architecture.

[67] Y. Ghigoff, J. Sopena, K. Lazri, A. Blin, G. Muller, BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing, in: 18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 21, USENIX Association, 2021, pp. 487–501, URL https://www.usenix.org/conference/nsdi21/presentation/ghigoff.

[68] G. Bertin, XDP in practice: integrating XDP into our DDoS mitigation pipeline, in: Technical Conference on Linux Networking, Netdev, Vol. 2, 2017.

[69] F. Incubator, Katran - A high performance layer 4 load balancer, On-line Git Repository URL https://github.com/facebookincubator/katran.

[70] J. Koch, M. Spier, B. Gregg, E. Hunter, Extending vector with eBPF to inspect host and container performance, 2019, URL https://medium.com/netflix-techblog/extending-vector-with-ebpf-to-inspect-host-andcontainer-performance-5da3af4c584b.

[71] D. Beckett, J. Joubert, S. Horman, Host dataplane acceleration (HDA), 2018.

[72] T. Braun, C. Diot, A. Hoglander, V. Roca, An experimental user level implementation of TCP, INRIA, RR-2650 (1995) URL https://hal.inria.fr/inria-00074040/document.

[73] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, Z. Shi, The QUIC transport protocol, in: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, ACM, 2017, http://dx.doi.org/10.1145/3098822.3098842.

[74] A.T.d. Oliveira Filho, Uma análise experimental de desempenho do protocolo QUIC, Dissertação (Mestrado) - UFPE, UFPE, Recife, 2020, p. 101, URL https://repositorio.ufpe.br/handle/123456789/37646.

[75] A.M. Kakhki, S. Jero, D. Choffnes, C. Nita-Rotaru, A. Mislove, Taking a long look at QUIC, in: Proceedings of the 2017 Internet Measurement Conference, ACM, 2017, http://dx.doi.org/10.1145/3131365.3131368.

[76] S. Cook, B. Mathieu, P. Truong, I. Hamchaoui, QUIC: Better for what and for whom? in: 2017 IEEE International Conference on Communications, ICC, IEEE, 2017, http://dx.doi.org/10.1109/icc.2017.7997281.

[77] X. Yang, L. Eggert, J. Ott, S. Uhlig, Z. Sun, G. Antichi, Making QUIC quicker with NIC offload, in: Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC, ACM, 2020, http://dx.doi.org/10.1145/3405796.3405827.

[78] K.L. McMillan, L.D. Zuck, Formal specification and testing of QUIC, in: Proceedings of the ACM Special Interest Group on Data Communication, ACM, 2019, http://dx.doi.org/10.1145/3341302.3342087.

[79] I. Marinos, R.N. Watson, M. Handley, Network stack specialization for performance, SIGCOMM Comput. Commun. Rev. 44 (4) (2014) 175–186, http://dx.doi.org/10.1145/2740070.2626311.

[80] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, E. Bugnion, IX: A protected dataplane operating system for high throughput and low latency, in: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI '14, USENIX Association, USA, 2014, pp. 49–65.

[81] A. Belay, G. Prekas, M. Primorac, A. Klimovic, S. Grossman, C. Kozyrakis, E. Bugnion, The IX operating system, ACM Trans. Comput. Syst. 34 (4) (2017) 1–39, http://dx.doi.org/10.1145/2997641.

[82] R. Dantas, D. Sadok, C. Flinta, A. Johnsson, KVM virtualization impact on active round-trip time measurements, in: 2015 IFIP/IEEE International Symposium on Integrated Network Management, IM, 2015, pp. 810–813, http://dx.doi.org/10.1109/INM.2015.7140382.

[83] G. Wang, T.S.E. Ng, The impact of virtualization on network performance of Amazon EC2 data center, in: 2010 Proceedings IEEE INFOCOM, 2010, pp. 1–9, http://dx.doi.org/10.1109/INFCOM.2010.5461931.

[84] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, J. Wiegert, Intel virtualization technology for directed I/O, Intel Technol. J. 10 (3) (2006).

[85] I. AMD, O. Virtualization, Technology (IOMMU) specification, 2007.

[86] A. Richter, C. Herber, T. Wild, A. Herkersdorf, Denial-of-service attacks on PCI passthrough devices: Demonstrating the impact on network- and storage-I/O performance, J. Syst. Archit. 61 (10) (2015) 592–599, http://dx.doi.org/10.1016/j.sysarc.2015.07.003.

[87] H.D. Chirammal, P. Mukhedkar, A. Vettathu, Mastering KVM Virtualization, Packt Publishing Ltd, 2016.

[88] PCI-SIG, Single-Root I/O Virtualization Specification, Internet URL http://www.pcisig.com/specifications/iov/.

[89] J. Zhang, X. Lu, D.K. Panda, Performance characterization of hypervisor-and container-based virtualization for HPC on SR-IOV enabled InfiniBand clusters, in: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW, IEEE, 2016, http://dx.doi.org/10.1109/ipdpsw.2016.178.

[90] P.S. Muhammad Siraj Rathore, KVM vs. LXC: Comparing performance and isolation of hardware-assisted virtual routers, Am. J. Netw. Commun. 2 (2013) 88–96, http://dx.doi.org/10.11648/j.ajnc.20130204.11.

[91] O. Open, Virtual I/O device (VIRTIO) specification, 2019, http://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.html.

[92] T.F.D. Project, FD.io - the universal dataplane, 2016, Internet URL https://fd.io/.

[93] J. Duman, FD.io VPP - Whitepaper, Cisco, 2017.

[94] T.F.D. Project, Use cases for FD.io, 2016, Internet URL https://fd.io/overview/usecases/.

[95] J. Tollet, VPP features, 2016, URL https://www.youtube.com/watch?v=djPjPhZ4Vr8.

[96] D. Marion, VPP architecture, 2016, URL https://www.youtube.com/watch?v=d6Fn5cuLcVk.

[97] T.F.D. Project, Scalar vs vector packet processing, 2016, Internet URL https://s3-docs.fd.io/vpp/22.02/aboutvpp/scalar-vs-vector-packet-processing.html.

[98] O. Hohlfeld, J. Krude, J.H. Reelfs, J. Rüth, K. Wehrle, Demystifying the performance of XDP BPF, in: 2019 IEEE Conference on Network Softwarization, NetSoft, 2019, pp. 208–212, http://dx.doi.org/10.1109/NETSOFT.2019.8806651.

[99] G. Sabin, M. Rashti, Security offload using the smartnic, a programmable 10 gbps ethernet NIC, in: 2015 National Aerospace and Electronics Conference, NAECON, 2015, pp. 273–276, http://dx.doi.org/10.1109/NAECON.2015.7443082.

[100] D. Raumer, S. Gallenmüller, P. Emmerich, L. Märdian, G. Carle, Efficient serving of VPN endpoints on COTS server hardware, in: 2016 5th IEEE International Conference on Cloud Networking, Cloudnet, 2016, pp. 164–169, http://dx.doi.org/10.1109/CloudNet.2016.25.

[101] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, L. Zhang, KV-direct: High-performance in-memory key-value store with programmable NIC, in: Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 137–152, http://dx.doi.org/10.1145/3132747.3132756.

[102] R. Neugebauer, G. Antichi, J.F. Zazo, Y. Audzevich, S. López-Buedo, A.W. Moore, Understanding PCIe performance for end host networking, in: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 327–341, http://dx.doi.org/10.1145/3230543.3230560.

[103] W. Sun, R. Ricci, Fast and flexible: Parallel packet processing with GPUs and click, in: Architectures for Networking and Communications Systems, IEEE, 2013, pp. 25–35.

[104] S. Bhattacharya, P.K.R. Maddikunta, R. Kaluri, S. Singh, T.R. Gadekallu, M. Alazab, U. Tariq, et al., A novel PCA-firefly based xgboost classification model for intrusion detection in networks using GPU, Electronics 9 (2) (2020) 219.

[105] P.B. Viegas, A.G. de Castro, A.F. Lorenzon, F.D. Rossi, M.C. Luizelli, The actual cost of programmable smartnics: Diving into the existing limits, in: L. Barolli, I. Woungang, T. Enokido (Eds.), Advanced Information Networking and Applications, Springer International Publishing, Cham, 2021, pp. 181–194.

[106] D. Sidler, G. Alonso, M. Blott, K. Karras, K. Vissers, R. Carley, Scalable 10gbps TCP/IP stack architecture for reconfigurable hardware, in: 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, IEEE, 2015, http://dx.doi.org/10.1109/fccm.2015.12.

[107] K. Fall, G. Iannaccone, M. Manesh, S. Ratnasamy, K. Argyraki, M. Dobrescu, N. Egi, RouteBricks: Enabling general purpose network infrastructure, SIGOPS Oper. Syst. Rev. 45 (1) (2011) 112–125, http://dx.doi.org/10.1145/1945023.1945037.

[108] M. Wu, Q. Chen, J. Wang, Toward low CPU usage and efficient DPDK communication in a cluster, J. Supercomput. (2021) http://dx.doi.org/10.1007/s11227-021-03942-x.

[109] M. Bertrone, S. Miano, F. Risso, M. Tumolo, Accelerating linux security with eBPF iptables, 2018, pp. 108–110, http://dx.doi.org/10.1145/3234200.3234228,

[110] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, G. Carle, Moongen: A scriptable high-speed packet generator, in: Proceedings of the 2015 Internet Measurement Conference, IMC '15, Association for Computing Machinery, New York, NY, USA, 2015, pp. 275–287, http://dx.doi.org/10.1145/2815675.2815692.

[111] J. Liu, Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support, in: 2010 IEEE International Symposium on Parallel Distributed Processing, IPDPS, 2010, pp. 1–12, http://dx.doi.org/10.1109/IPDPS.2010.5470365.

[112] A. Richter, C. Herber, H. Rauchfuss, T. Wild, A. Herkersdorf, Performance isolation exposure in virtualized platforms with PCI passthrough I/O sharing, in: Architecture of Computing Systems – ARCS 2014, Springer International Publishing, 2014, pp. 171–182, http://dx.doi.org/10.1007/978-3-319-04891-8_15.

[113] X. Xu, B. Davda, SRVM: Hypervisor support for live migration with passthrough SR-IOV network devices, SIGPLAN Not. 51 (7) (2016) 65–77, http://dx.doi.org/10.1145/3007611.2892256.

[114] E. Freitas, A.T.d. Oliveira Filho, P.R.X.d. Carmo, D.F.H. Sadok, J. Kelner, Takeaways from an experimental evaluation of express data path (XDP) and data plane development kit (DPDK) under a cloud computing environment, Res. Soc. Dev. 11 (12) (2022) http://dx.doi.org/10.33448/rsd-v11i12.34200.

[115] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, et al., The design and implementation of open $\{vSwitch\}$, in: 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, 2015, pp. 117–130.

[116] R.H.C. Portal, Deploying mobile networks using network functions virtualization - performance and optimization, 2017, Internet URL https://access.redhat.com/documentation/en-us/reference_architectures/2017/html/deploying_mobile_networks_using_network_functions_virtualization/performance_and_optimization.

[117] N. Pitaev, M. Falkner, A. Leivadeas, I. Lambadaris, Characterizing the performance of concurrent virtualized network functions with OVS-DPDK, fd.io VPP and SR-IOV, in: Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 285–292, http://dx.doi.org/10.1145/3184407.3184437.

[118] R. Kawashima, S. Muramatsu, H. Nakayama, T. Hayashi, H. Matsuo, A host-based performance comparison of 40g NFV environments focusing on packet processing architectures and virtual switches, in: 2016 Fifth European Workshop on Software-Defined Networks, EWSDN, 2016, pp. 19–24, http://dx.doi.org/10.1109/EWSDN.2016.11.

[119] R. Kawashima, H. Nakayama, T. Hayashi, H. Matsuo, Evaluation of forwarding efficiency in NFV-nodes toward predictable service chain performance, IEEE Trans. Netw. Serv. Manag. 14 (4) (2017) 920–933, http://dx.doi.org/10.1109/tnsm.2017.2734560.

[120] S. Shanmugalingam, A. Ksentini, P. Bertin, DPDK open vswitch performance validation with mirroring feature, in: 2016 23rd International Conference on Telecommunications, ICT, IEEE, 2016, http://dx.doi.org/10.1109/ict.2016.7500387.

[121] J. Tan, C. Liang, H. Xie, Q. Xu, J. Hu, H. Zhu, Y. Liu, VIRTIO-USer, in: Proceedings of the Workshop on Kernel-Bypass Networks, ACM, 2017, http://dx.doi.org/10.1145/3098583.3098586.

[122] M.-A. Kourtis, G. Xilouris, V. Riccobene, M.J. McGrath, G. Petralia, H. Koumaras, G. Gardikis, F. Liberal, Enhancing VNF performance by exploiting SR-IOV and DPDK packet processing acceleration, in: 2015 IEEE Conference on Network Function Virtualization and Software Defined Network, NFV-SDN, 2015, pp. 74–78, http://dx.doi.org/10.1109/NFV-SDN.2015.7387409.

[123] P. Zhang, Configuring and Benchmarking Open vSwitch, DPDK and vhost-user, Tech. rep., KVM Forum, 2017, URL http://events17.linuxfoundation.org/sites/events/files/slides/Configuring%20and%20Benchmarking%20%20Open%20vSwitch%2C%20DPDK%20and%20vhost-user_Pei%20Zhang.pdf.

[124] J. Hu, Accelerating Packet Processing with GRO and GSO in DPDK, Tech. rep., DPDK Summit, 2017, URL https://www.dpdk.org/wp-content/uploads/sites/35/2018/06/GRO-GSO-Libraries-Bring-Significant-Performance-Gains-to-DPDK-based-Applications.pdf.

[125] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, D. Walker, P4: Programming protocol-independent packet processors, SIGCOMM Comput. Commun. Rev. 44 (3) (2014) 87–95, http://dx.doi.org/10.1145/2656877.2656890.

[126] S. Arslan, S. Ibanez, A. Mallery, C. Kim, N. McKeown, Nanotransport: A low-latency, programmable transport layer for NICs, in: Proceedings of the ACM SIGCOMM Symposium on SDN Research, SOSR,SOSR '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 13–26, http://dx.doi.org/10.1145/3482898.3483365.

[127] D. Carrascal, E. Rojas, J. Alvarez-Horcajo, D. Lopez-Pajares, I. Martínez-Yelmo, Analysis of P4 and XDP for IoT programmability in 6G and beyond, IoT 1 (2) (2020) 605–622, http://dx.doi.org/10.3390/iot1020031.

[128] W. Tu, F. Ruffy, M. Budiu, Linux network programming with P4, in: Linux Plumbers' Conference 2018, 2018, URL http://vger.kernel.org/lpc_net2018_talks/p4-xdp-lpc18-paper.pdf.

[129] VMWare, P4C-XDP - Backend for the P4 compiler targeting XDP, in: n-line Git Repository URL https://github.com/vmware/p4c-xdp.