Fast Packet Processing: A Survey

Danilo Cerović, *Graduate Student Member, IEEE*, Valentin Del Piccolo, Ahmed Amamou, *Member, IEEE*, Kamel Haddadou, *Member, IEEE*, and Guy Pujolle, *Senior Member, IEEE*

***Abstract*—The exponential growth of data traffic, which is not expected to stop anytime soon, brought about a vast amount of advancements in the networking field. Latest network interfaces support data rates in the range of 40 Gb/s and higher. This, however, does not guarantee higher packet processing speeds which are limited due to the overheads imposed by the architecture of the network stack. Nevertheless, there is a great need for a speedup in the forwarding engine, which is the most important part of a high-speed router. For this reason, many software-based and hardware-based solutions have emerged recently with a goal of increasing packet processing speeds. An operating system's networking stack is conceived for general purpose communications rather than high-speed networking applications. In this paper, we investigate multiple approaches that attempt to improve packet processing performance on server-class network hosts, either by using software, hardware, or the combination of the two. We survey various solutions, among which some are based on the Click modular router, which offloads its functions on different types of hardware like graphics processing units, field programmable gate arrays or different cores among different servers with parallel execution. Furthermore, we explore other software solutions which are not based on the Click modular router. We compare them in terms of the domain in which they operate (user-space or kernel-space). Then we compare them based on their use of zero-copy techniques, batch packet processing, and parallelism. We also discuss different hardware solutions and compare them additionally in terms of the type of hardware that they use, their usage of CPU and how they connect with it. Furthermore, we discuss the integration possibilities in virtualized environments of the described solutions, their constraints and their requirements. At last, we discuss the latest approaches and the future directions in the field of fast packet processing.**

***Index Terms*—Computer networks, packet switching, TCPIP, next generation networking, network architecture, routing protocols, network function virtualization, field programmable gate arrays, data communication, switches.**

I. Introduction

PACKET processing relates to a large collection of algorithms which are applied to a packet of information or data which moves between different network elements. With the increased performance of network interfaces, there is a corresponding need for faster packet processing. The most important part of a high-speed router is the forwarding engine and with the exponential growth of Internet traffic, which is not expected to slow down [1], there is a high demand for the introduction of multi-terabit IP routers [2]. Similarly, inside the datacenters there is the rising need for switches that support throughputs that go from hundreds of Gbps to even multiple Tbps. Examples of such switches are Intel FM10000 series [3] and Broadcom Tomahawk 3 [4] and they are used to aggregate a large number of high-speed links. However, in this survey we investigate a parallel problematic which is fast packet processing solutions on server-class network hosts by using software, hardware or the combination of the two. These solutions allow building Virtual Network Functions (VNFs) without a need for a dedicated networking hardware. The trade-off is the performance gap that these solutions are constantly trying to reduce when compared with the dedicated hardware.

In today's datacenters, it is typical for servers to have network interfaces with speeds of 40 Gbps [5]. However, this does not guarantee a higher packet processing speed since there are various overheads which are due to network stacks' architectural design. The generality of network stack architecture comes with a high performance cost which immensely limits the effectiveness of increasingly powerful hardware [6]. The overheads that it implies represent a key obstacle to effective scaling. In fact, in a modern operating system, moving a packet between the wire and the application can take 10-20 times longer than the time needed to send a packet over a 10-gigabit interface [7].

A standard general-purpose kernel stack can hardly support multiple 10Gbps interface packet processing at line rate. For this reason, multiple software-based and hardware-based solutions have been proposed lately that aim to improve packet processing speed. Different user-space I/O frameworks allow bypassing the kernel and obtaining a batch of raw packets through a single syscall. They also add support for modern Network Interface Card (NIC) capabilities such as multi-queueing [8]. Namely, Intel's DPDK [9], Netmap [10], PacketShader [11], PF_RING [12], OpenOnload [13], PACKET_MMAP [14] are examples of such frameworks out of which most relevant ones, in terms of their widespread usage, are reviewed in this paper among other solutions.

Many solutions are based on the Click Modular Router [15] which represents a software architecture for building flexible and configurable routers. Some of the most notable ones are: RouteBricks [16], FastClick [8], Snap [17], DoubleClick [18], SMP Click [19], ClickNP [20], NP-Click [21]. Majority of

Manuscript received July 19, 2017; revised January 26, 2018 and May 2, 2018; accepted June 9, 2018. Date of publication June 27, 2018; date of current version November 19, 2018. This work was supported in part by FUI-APP19 CARP and in part by FUI-APP22 WOLPHIN2.0 Projects from Région Ile-de-France. *(Corresponding author: Danilo Cerović.)*

D. Cerović is with the Research and Development Department, GANDI SAS, 75013 Paris, France, and also with Sorbonne Université (UPMC—LIP6), 75005 Paris, France (e-mail: danilo.cerovic@gandi.net).

V. Del Piccolo, A. Amamou, and K. Haddadou are with the Research and Development Department, GANDI SAS, 75013 Paris, France (e-mail: valentin.d.p@gandi.net; ahmed@gandi.net; kamel@gandi.net).

G. Pujolle is with the Sorbonne Université (UPMC—LIP6), 75005 Paris, France (e-mail: guy.pujolle@lip6.fr).

Digital Object Identifier 10.1109/COMST.2018.2851072

these solutions aim to improve Click by allowing to parallelize the packet processing workload on multicore systems.

As has been discussed in [16] basically there are two possible approaches to building programmable network infrastructures with high performance. The first is to start from high-end specialized equipment and make it programmable such as the ideas of OpenFlow [22], [23]. The other would be to start from software routers and optimize their packet processing performance. This second approach would give programmers the opportunity to work in the well-known environments of general-purpose computers.

However, there are "semi-specialized" solutions such as the ones which use network processors to offload a part of the packet processing, which is a bottleneck, while the rest of the packet processing is done on conventional processors. Examples of solutions which use NPs (Network Processors) are NP-Click [21] along with works such as [24] in which the authors implement OpenFlow switch over the EZchip NP4 Network Processor. Chang and Kuo [25] investigate many programming techniques and optimization issues that can significantly improve the performance of Network Processors. Furthermore, [26] provides deep understanding of NP design trade offs by using an analytic performance model which is configured by realistic workload and technology parameters. On the other hand, there are solutions which use reconfigurable hardware (Field Programmable Gate Arrays - FPGAs) as they are more versatile compared to NPs. Namely, Chimpp [27], ClickNP [20], Netfpga [28], SwitchBlade [29], GRIP [30] along with other works in this area [31], [32]. The most well-known solutions which use Graphics Processing Units (GPUs) for packet processing are certainly PacketShader [11], Snap [17], APUNet [33] and GASPP [34].

The performance of each solution depends on whether the underlying hardware is a CPU, GPU or an FPGA. Each one has its own advantages and disadvantages. The GPUs exploit their extreme thread-level parallelism, well suited for packet processing applications (which can be parallelized). However, the CPUs introduce less latency than GPUs. Also, the programs written for CPUs can last longer (and generally execute faster on new CPU generations) which is not necessarily the case for GPUs because the CPUs have better API consistency than GPUs. The FPGAs are more energy efficient than GPUs and CPUs but they are not easily programmable as they use HDL (Hardware Description Language). Additionally, FPGAs provide deterministic timing with latencies that are one order of magnitude lower than GPUs [35]. This is very important for hard real-time tasks and applications like network synchronization, encryption etc. We will further discuss this topic in Section VI.

All of the mentioned solutions can be equally suitable for the virtualized environments as there is a great interest for a packet processing acceleration because of the inherent performance loss that these environments bring. The Netmap API can be adapted to be used as a communication mechanism between a hypervisor and a host as in the VALE [36] solution. Furthermore, in [37] the packet I/O speed in the VMs is further improved, while ptnetmap [38] is the Virtual Passthrough solution based on the netmap framework. There are also solutions which are based on DPDK that try to accelerate the packet processing speed for VMs. Namely, NetVM [39] is a platform built on top of the KVM and DPDK library and it is used for high-speed packet processing. Moreover, the DPDK can be used to boost the performance of the open source virtual switch (OvS [40]) as in OvS-DPDK [41] solution. OvS is a well-known example of an OpenFlow enabled switch. OpenFlow is a protocol which is used in SDN (Software Defined Networks) for the communication between a controller (which implements the control plane) and a switch (which in the case of SDN implements the data plane). The main purpose of OvS was not to accelerate the traffic but rather to allow the communication in multi-server virtualization deployments. The OvS-DPDK solution allows accelerating the packet processing of the OvS switch.

Another solution called OpenWrt [42] is used specifically for wireless routers. OpenWrt is a GNU/Linux distribution for embedded devices (wireless routers) which is used instead of static firmwares which do not have a fully writable filesystem and are not highly customizable [42]. This solution does not represent the fast packet processing solution but rather a (customizable) firmware replacement for the existing routers (list of the supported hardware is available on OpenWrt's website [43]).

The rest of this paper can be outlined as follows. In Section II we briefly present terminology that is commonly used for this topic and we provide background information related to data rates supported by communication links and the ability of systems to deal with the required packet processing speed. Section III introduces multiple software packet processing implementations where we start with Click-based solutions and then continue with software solutions which are not based on Click Modular router. Section IV discusses different hardware implementations of packet processing. In Section V we compare different software implementations by using 4 criteria (operations in user-/kernel-space, zero copy technique, support of batch processing and parallelism) and we discuss performance comparison of different IO frameworks. We present a comparison and discussion on hardware solutions in Section VI and afterward, in Section VII, we present the integration possibilities in virtualized environments. Discussion on latest approaches and future directions is given in Section VIII, and we conclude in Section IX.

II. Background

A. Terminology

All the procedures performed by the router can be divided into two categories: *time-critical* and *non-time critical* which depends on how often these operations are performed [44]. *Time-critical* operations are referred to as *fast path* and represent the tasks which are performed on the majority of packets that pass through the router. It is important that these procedures are performed in an optimized way in order to reach gigabit data rates. On the other hand, *non-time critical* operations are referred to as *slow path*. They are carried out mostly on packets which are used for management, error handling and maintenance. *Slow path* operations are implemented in

such a way that they never interfere with *fast path* operations as the *fast path* always has the higher priority. In the rest of this section we will define the terms *Fast Path* and *Slow Path* from the router architecture point of view. However, we should note here that the two terms can be also used to describe the type of solution in question. Namely, the *fast path* solution represents the mechanism which can accelerate the packet processing speed. For example, in some architectures the *fast path* (solution) is used to forward the high-priority packets at wire speed with minimal packet latency, while the *slow path* (e.g., the OS networking stack) will forward the low-priority packets. *Fast path* architecture generally includes a reduced number of instructions or particular optimization methods when compared to the *slow path*. In packet processing systems, the *slow path* is typically run on the operating system's networking stack while the *fast path* often optimizes the existing networking stack or performs hardware acceleration in order to avoid overheads that occur in the networking stack.

1) Fast Path: Time-critical operations can be classified into two groups [44]: *forwarding* and *header processing*. Forwarding tasks include packet classification, service differentiation and destination address lookup operation, while header processing tasks include checksum calculation, packet validation and packet lifetime control. All these *fast path* operations in high-performance routers are implemented in the hardware. This is due to the fact that these operations need to be executed in real time for each individual packet. As it can be seen on Figure 1, the packet which travels on the *fast path* is only processed by the modules on the line cards and does not go on the route processor card.

The packets on the *fast path* are processed and transferred from the ingress line card to the egress line card through the backplane. Typically, ASICs are used for the implementation of fast path functions in order to achieve high throughput.

In the header processing part of the fast path, each IP packet that enters the router passes an array of validity checks. This is done in order to verify if the packet should be discarded or not. The first verification is whether the version of IP packet is properly set (IPv4 or IPv6) or not. Afterward comes the check of the length of the IP packet reported by MAC or link layer, followed by a verification that the calculated checksum of the IP packet is equal to the one written in the IP header. Then the TTL field gets decremented and the new checksum for the IP packet is calculated.

Packet forwarding consists of finding the appropriate egress network interface of the router to which a packet needs to be forwarded and finding the appropriate MAC to which the packet is going to be forwarded. This is done by performing a lookup in the forwarding table.

Packet classification is a process of identifying and mapping different types of IP traffic, based on the information contained in the packet, to a suitable action according to a sequence of rules [44], [45]. Packets can be differentiated by the source IP address, destination IP address, source port, destination port and protocol flags. This group is usually called a 5-*tuple*.

In a situation where multiple packets need to be forwarded to a single port of the router at the same time, packet queueing and packet scheduling techniques are used. First of all, every packet is put in a queue. Then, according to the scheduling algorithm and based on the class of the packet and the service guarantees associated to it, the decision is made about which packet should be sent next.

2) Slow Path: Packets that pass by the slow path are processed to a certain extent by the line card before getting sent to the CPU for further processing, as can be seen in Figure 1. In the *slow path*, there are *non-time-critical* operations which include [44]:

- Processing of data packets that lead to errors in the *fast path* and informing the originating source of the packets by sending ICMP packets
- Processing of keep-alive messages of the routing protocol from the adjacent neighbors and sending such messages to the neighboring routers
- Processing of incoming packets which contain route table updates and informing adjacent routers on each modification in network topology
- Processing of packets related to management protocols such as SNMP

Address Resolution Protocol (ARP) processing is one of the *slow path* functions which is used for finding the layer 2 address when forwarding the packet. In fact, if the router knows the destination IP address of the next hop, it needs to determine the corresponding link-level address before sending the packet on the router interface. When the packet needs to be forwarded and the layer 2 address of the destination is unknown, ARP is used, since these link-level addresses are obtained during the process of address lookup operation. For this reason, some network designers implement the ARP on a *fast path*, but others implement them on a *slow path* since those requests do not appear very frequently.

When a Maximum Transmission Unit (MTU) is not the same on different interfaces of a router, the packet fragmentation and reassembly can be performed [44]. This is done in order to be able to send packets from one interface to some other interface which has an MTU size smaller than the size of the packet. A disadvantage of doing fragmentation is the added complexity of the router structure and a reduction of data throughput since an IP packet needs to be retransmitted each time when a fragment is lost. Also, for the same throughput more headers are transferred and consequently, less data is transferred. Since path MTU discovery is often used for discovering the smallest MTU size on the path before packet transmission, there is no need to implement fragmentation function in the *fast path*. The fragmentation of IP packets is different for the IPv4 and the IPv6. For the IPv4, whenever a router receives a PDU (Protocol Data Unit) which is larger than the next hop's MTU, it either drops the PDU (and sends the corresponding ICMP message) or it fragments the IP packet in order to be able to send it according to the MTU size. For the IPv6, the two endpoint hosts of a communication session determine the optimal path MTU before they start exchanging packets. The IPv6 routers do not perform fragmentation, but they drop the packets which are larger than the MTU.

There are also some advanced IP options like source routing, time stamping, route recording and ICMP error generation, but since these functions are used quite rarely, it is enough to implement them in the control processor in the *slow path*.

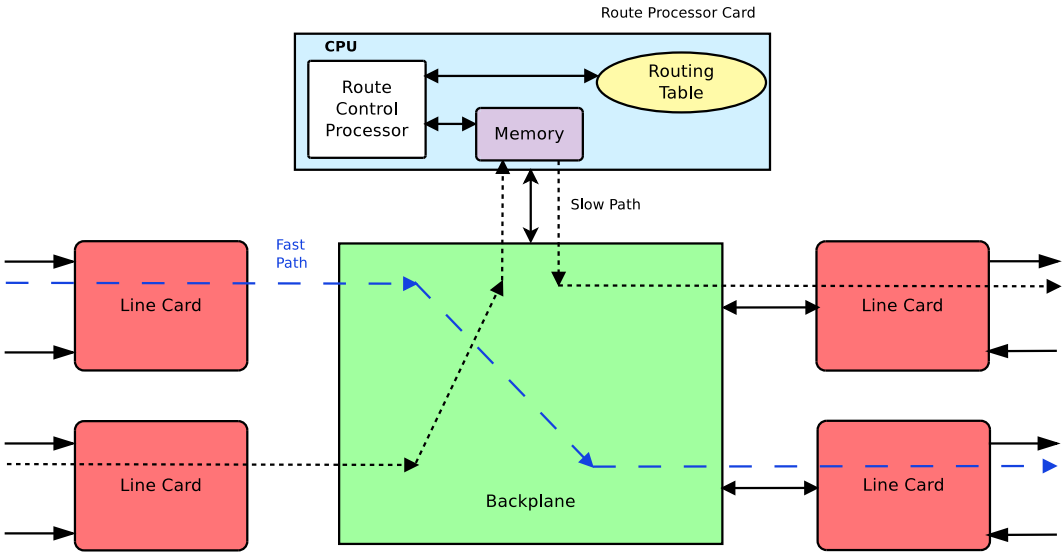

Fig. 1. Components of a router [44].

B. Background on Packet Processing

In computer networking, historically there has always been a contest between the data rate supported by the communication link and the ability of a system to deal effectively with the required packet processing speed. They both had the same target of maximizing the utilization of available resources and providing the fastest possible service. It should be noted that packet processing overhead comes on a per-packet basis and consequently, for certain throughput, smaller packets produce higher overhead.

Supporting a high throughput implicates a sufficiently low packet processing latency (or delay). For example, in order to be able to support forwarding of the incoming traffic of a certain throughput by some packet processing solution, this solution needs to have sufficiently low packet processing delay. Otherwise, packet arrival speed will be higher than the packet departure speed which will certainly cause the overflow of a queue (buffer) which receives the packets. This means that if a mean process delay per packet is longer than the packet inter-arrival time, then the solution in question will not support the incoming throughput because the buffer will get overloaded with time and the packets will be dropped. For this reason, as described in the following sections, many packet processing solutions process packets in batches which allows taking more time to receive multiple packets and process them in parallel. When sending a packet from one node to another, there are 4 types of delays [46]: transmission delay (time needed to send a packet onto the wire), propagation delay (time needed to transmit a packet via a wire), processing delay (time needed to handle the packet on the network system) and the queuing delay (time during which the packet is buffered before being sent).

Five main steps of software packet processing are [47]: 1) packet reception from a NIC and its transfer to the memory with DMA (Direct Memory Access), 2) packet processing by CPU, 3) transfer of processed packet to the ring buffer of a target NIC, 4) writing the tail of the ring buffer to the NIC register, 5) packet transfer to the NIC with DMA and further from the NIC.

If used in the right way, multi-core architectures can bring significant improvements in packet processing speeds. However, it is not enough just to partition the tasks to multiple concurrent execution threads since general-purpose operating systems provide mechanisms which are not suitable for high performance in packet forwarding. In order to use multi-core processors more efficiently, multiqueue NICs have been introduced. Network card manufacturers have modified the design of network adapters as they logically partition them into multiple independent RX/TX queues [48]. For instance, each RX queue is assigned to a separate interrupt, and each interrupt is routed to different core. In this way, traffic from a single NIC can be distributed among different cores. The number of queues is usually limited to the number of available processor cores [48]. Receive-Side Scaling (RSS) [49] is a mechanism which enables distributing packet processing workload among different cores. Besides all these improvements, individual queues are not exposed to applications and operations need to be serialized since all threads are accessing the same Ethernet device.

Even though network stack and network drivers are very flexible as they allow new protocols to be implemented as kernel modules, packet forwarding goes through many software layers from packet reception to packet transmission which leads to performance bottlenecks. Higher levels of the operating system do not have full hardware control and consequently, they add up cycles. In addition, locks or mutexes (in multicore environments) and context switching use even more core cycles.

Network device drivers have been created mainly for general purpose packet processing. For each received packet, the operating system allocates a memory in which packets are queued and sent to network stack. After the packet has been sent to the network stack, this memory is freed. Packets are first copied to the kernel, in order to be transmitted to a user-space which

increases the time of the packet's journey. For the purpose of reducing the packet journey several packet capture solutions implement memory-mapping techniques which reduces the cost of packet transmission from kernel-space to user-space through system calls [50]. On the other hand, packets waiting for transmission are first put to memory on network adapters before transmission over the wire. Two circular buffers, one for transmission and one for reception are allocated by network device drivers.

In order to accelerate the packet processing speed and offload (a part of) packet processing from the CPUs, some of the packet processing solutions that are presented in the following sections use a specific type of hardware like GPUs and FPGAs. The GPUs offer extreme thread level parallelism while the CPUs maximize instruction-level parallelism. In general, GPUs are very well suited for packet processing applications as they offer data-parallel execution model. They can possess thousands of processing cores and they adopt single-instruction, multiple thread (SIMT) execution model where a group of threads (called waveforms in AMD and warp in NVIDIA hardware) execute concurrently [33]. Similarly, the FPGAs have a massive amount of parallelism built-in because they posses millions of Logic Elements (LEs) and thousands of DSP blocks. However, they have an increased programming complexity which is the main challenge for using an FPGA as an accelerator. The FPGAs are programmed with HDLs (Hardware Description Languages) such as Verilog and VHDL. Their programming, debugging and productivity difficulties are not negligible. However, there are HLS (High-Level Synthesis) tools that try to overcome this problem by allowing to program FPGAs in high-level programming languages.

III. Software Implementations

In this section, we introduce the 7 different free software implementations that are the most well known in this area of research, that have the widespread use, or are highly performant. These solutions can be used as either a part of, or as a complete architecture of a fast packet processing solution. We start with the Click Modular Router along with Click-based solutions such as RouteBricks and FastClick. Additionally, we describe Netmap, NetSlices, PF_RING and DPDK solutions. Other solutions that are based on Click Modular Router and that offload part of the processing on a specialized hardware are represented in the next section.

A. Click-Based Solutions

1) Click: Click [15] introduced one of the first modular software architectures used to create routers. Previously, routers had inflexible closed designs so it wasn't straightforward for network administrators to identify the interactions between different router functions. For the same reason, it was very difficult to extend the existing functions. The answer to such problems lies in the idea of building a router configuration from a collection of building blocks or fine-grained components which are called *elements*. Each building block represents a packet processing module. The connection of the certain collection of *elements* forms directed graph. Thus, a router configuration can be extended by writing new *elements* or modifying existing ones.

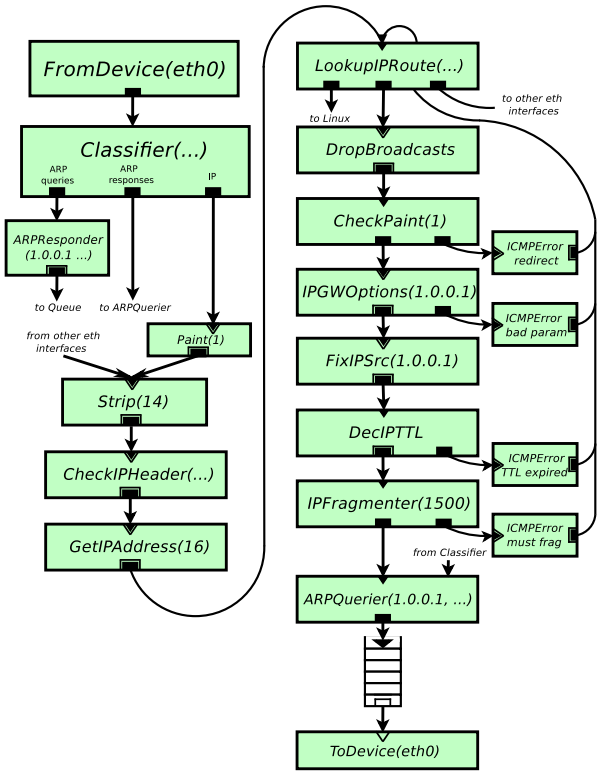

Fig. 2. Example of a Click IP router configuration [15].

In Figure 2, an example of the IP router configuration is shown. The vertices of the directed graph represent *elements* in the Click router's configuration and each edge represents a *connection* and it is actually a possible path on which a packet might be transferred. The scheme has been simplified to show only one interface (*eth0*), but it can be extended with the appropriate elements of the Click router to represent the case of having multiple network interfaces. Each *element* in Figure 2 has its own functionality.

First of all, the *FromDevice* element polls the receive DMA queue of its device and if it finds the newly arrived packets, it pushes them through the configuration [15]. Afterward, the *Classifier* element classifies Ethernet packets into ARP queries, ARP responses and IP packets. Then, it forwards them to the appropriate output port accordingly. The *ARPResponder* replies on ARP queries by sending ARP responses. The *Paint* element marks packets with an "integer" color which allows the IP router to decide whether a packet is sent over the same interface from which it was received [15]. Next, the *Strip* element strips the specified number of bytes from the beginning of the packet. After that, the *CheckIPHeader* block verifies if the properties of the header conform to the standard, like the IP source address, the IP length fields and the IP checksum [15]. Then, the *GetIPAddress* element copies an address from the IP header into the annotation that represents the piece of information attached to the packet header. This information does not make part of the packet data [15]. Moreover, the *LookupIPRoute* element looks up the routing

table by taking the IP address from the annotation in order to find the appropriate output port. The *DropBroadcasts* element drops the packets that arrived as link-level broadcasts [15]. Next, the *CheckPaint* element forwards a packet on its first output. If the packet has a specific paint annotation it also forwards the copy of the packet to its second output. Afterward, the *IPGWOptions* element processes IP Record Route and Timestamp options of the packet [15]. If the packet has invalid options, it is sent to the second port. The *FixIPSrc* element checks if the packets' Fix IP Source annotation is set. In that case, it sets the IP header's source address field to the static IP address *ip* [15]. Otherwise, it forwards the packet unchanged. Then, the *DecIPTTL* element decrements the IP packet's time to live field and the *IPFragmenter* fragments IP packets that are larger than the specified MTU size. The *ARPQuerier* uses the ARP protocol to find the Ethernet address that corresponds to the destination IP address annotation of each input packet [15]. Finally, the *ToDevice* element sends packets to a Linux device driver for transmission.

Each *element* is implemented as a C++ object which may maintain private states, while the *connections* are implemented as pointers to *element* objects. The essential properties of an *element* are:

- *Element Class* - each of the elements belongs to one element class which defines the initialization procedure along with the part of the code which should be executed when the packet is processed by the element.
- *Ports* - each *element* can have an arbitrary number of ports and each output port of one *element* is connected to the corresponding input port of the other element.
- *Configuration String* - optional string used to pass additional arguments to the element at the initialization of the router.
- *Method Interfaces* - one or more method interfaces can be supported by each *element*, either for packet transfer or for exporting some other relevant data like queue length.

Connections in a Click router can be one of two types: push or pull. Between two push ports there is a push connection and between two pull ports there is a pull connection. The difference between the two types of connections is that in a push connection, it is the source *element* which initiates the packet transfer, while in the case of a pull connection, the transfer initiator is the destination *element*. There are also agnostic ports which can act as push ports when connected to push ports or pull ports when connected to pull ports.

All the queues are implemented as a separate Queue element which allows explicit control on how the packets are stored.

Click uses very simple configuration language which consists of two essential constructs: *declarations* and *connections*. *Declarations* create *elements*, while *connections* describe how they should be connected. Both user-level driver and Linux in-kernel driver can run Click router configurations.

This router is modular and therefore supports many extensions like Differentiated Services, queueing, dropping, scheduling policies, etc. So it is possible to build either simple or complex packet processors by using a subset of existing elements or extending it with new functionalities.

Even though the original Click system shared the Linux interrupt structure and device handling, since these two took up most of the time needed to forward a packet, it was decided that the polling should be used instead of the interrupts.

2) RouteBricks: RouteBricks [16] is a software router architecture which can run on multiple cores of a single server, or on multiple interconnected servers in order to leverage the performance of a software router. The idea behind RouteBricks is to start from a software router and try to optimize its packet-processing performance in order to attain speeds which are comparable to standard routers.

Parallelizing across servers is used for dividing packet processing tasks among different servers and for their engagement in cluster-wide distributed switching. The following guarantees are required: 100% throughput (which means providing full line rate of R bps, if needed), fairness where each input port can get its fair share of the capacity of any of the output ports and avoiding packet reordering. However, certain limitations are imposed by commodity servers: limited rates of internal links (they can not be higher than the external line rate R), limited per-node processing rate and limited per node fanout.

Direct VLB (Valiant Load Balancing) is a load-balancing routing algorithm that is used by RouteBricks. The architecture presented in [16] relies on two key assumptions. The first assumption about RouteBricks router is that it can handle at least one router port of rate R [16]. A minimum rate at which the server needs to process packets is $2R$ in the case of uniform traffic matrix, or $3R$ in the worst case traffic matrix. The second assumption is that a practical server-based VLB implementation works inside theoretical limits such as per-server processing rate of $2R$-$3R$.

When performing parallelism within servers, multiple cores are used at the same time in order to increase the performance of the system. In [16] it has been shown that using multiple cores is not sufficient for attaining rates comparable with standard routers. In the first case, Xeon servers (with eight 2.4GHz cores in total) are used in a traditional shared bus architecture where the communication between the sockets, memory and I/O devices is routed over the shared front-side bus with the single external memory controller. In the second case, where the Nehalem [51] servers (with eight 2.8GHz cores in total) are used, this communication is done over a mesh of dedicated point-to-point links. Multiple memory controllers are used, each integrated within sockets. It turns out that the first architecture has much lower aggregate memory bandwidth. In an earlier study done by the same authors [52] they found that the bottleneck is the shared bus which connects the CPU to the memory subsystem. This is why Nehalem architecture is used in RouteBricks, which implements parallelism at the CPUs coupled with parallelism in memory access.

Multi-queue NICs are used in order to satisfy multiple requirements imposed when workload needs to be distributed among available cores. In fact, whenever a receive or transmit queue is accessed by multiple cores, each queue needs to be locked before access which is a problem in the case of a large number of packets. This is why each queue needs to be accessed by a single core. Secondly, in [16] it has been shown

that each packet needs to be handled by a single core. As a matter of fact, they have shown that processing a packet in the pipeline across different cores compared to parallel processing (where each core does all the packet processing) gives significantly lower throughput.

Batch processing is also important since it reduces the number of transactions on the PCIe and I/O buses and improves the performance by the factor of 6 for the configuration implemented in RouteBricks.

In performance evaluation, for different packet sizes, three types of applications are used. One is minimal forwarding which does not use routing table when forwarding traffic from port *i* to port *j*. The other two are IP routing and IPsec packet encryption. For every major system component, the upper bound on the per-packet load has been estimated as a function of the input packet rate. Major system components that have been tested are: CPUs, memory buses, socket-I/O links, the inter-socket link and the PCI-E buses connecting the NICs to the I/O hub. This allowed comparing actual loads to upper bounds and show which components are likely to be bottlenecks. It has been shown that for all three applications, CPU load reaches the nominal upper bound which means that the system bottleneck is CPU [16]. However, as the number of cores per CPU tends to increase over the years, it goes along with the need for the software routers. The results show that in the worst case where packet size is 64B, the throughput of a single server for minimal packet forwarding is 9.7Gbps, while for IP routing it is 6.35Gbps. A parallel router, which consists of 4 Nehalem servers which are connected in the mesh topology, is named RB4. This router manages to keep with the given workload at 35Gbps where each NIC performs near its limit of 12Gbps. In fact, at 35Gbps, each NIC handles approximately 8.75Gbps of traffic on its external port and approximately 3Gbps on its internal port.

3) FastClick: FastClick [8] is a solution which integrates both DPDK and Netmap in Click. In fact, the authors developed two versions of FastClick. The first one uses DPDK and the other one uses Netmap. For this reason, the FastClick can be considered as a *fast path* version of Click router in the way that it exploits the advantages of DPDK and Netmap in order to accelerate Click's packet processing speed.

The choice of DPDK is based on the fact that it seems to be the fastest solution in terms of I/O throughput [8]. Whereas Netmap is used for its fine-grained control of both RX and TX rings. Additionally, Netmap has already been implemented for Click which provides the authors a base for their work.

These solutions enhance Click by providing the following features: I/O batching, computation batching, multi-queue support, and zero-copy.

The I/O batching is commonly used in packet processing frameworks as it allows to amortize the high per-packet overhead over the several packets. The costs of common tasks per each packet are reduced. For instance, the system call cost is amortized, while the instruction cache locality, the prefetching effectiveness and the prediction accuracy are improved [53]. Also the CPU consumption on per-packet memory management is eliminated [18]. The I/O batching is achieved in FastClick thanks to both DPDK and Netmap which can receive and send packets in batches.

For the computation batching, which is passing batches of packets between click elements and having each element working on the batches of packets instead of only one packet, the authors used a linked list and the click packets-annotation in order to stay Click compatible. Additionally, they chose the size of the computation batch as the total number of packets in the receive ring, therefore, the size evolves dynamically. This choice for the size of the compute batch reduces the latency of packets as the output operation will be done each time it receives a batch of packets instead of waiting for a certain number of packets.

The batching of both I/O and computation improve throughput for both versions of FastClick (DPDK and Netmap).

The support of multi-queue is achieved in different ways depending on the version of Fastclick used. The DPDK version can use a different number of queues in RX and TX whereas the Netmap version can not. This results in having opposite results for both versions. The DPDK version can have multiple TX queues and only one RX queue which improves its throughput. Whereas the Netmap version has a deteriorated throughput, having multiple RX queues forces the application to check every queue before processing the packets thus, inducing latency.

The management of the zero-copy feature is also achieved differently based on the Fastclick version. The Netmap version possesses the ability to swap buffers from the transmit and receive rings. This means that when receiving a packet at the NIC, the packet is written in a buffer in the receiving ring and this buffer can then be swapped with an empty one in order to keep the receiving ring empty. By doing this swap of a buffer, there has been zero copy of the packet. For the DPDK version, DPDK already provides a swapped buffer which allows sending the received packet directly to the transmit ring instead of copying it. Therefore, the packet is never copied. Using this zero-copy feature in both versions increases their throughput.

In addition to enhancing Click with these four features, the authors made some tests to verify if they could improve even more the throughput of FastClick. First, they checked the ring size of both the RX and TX queue. However, they found that this parameter does not significantly influence the performance of FastClick.

Secondly, they modified the execution model. Instead of using a FIFO queue to store the incoming packet, they choose a "full-push" model without queue and in which the packets traverse the forwarding path without interruption and they are managed by a unique thread.

In conclusion, the FastClick solution is a software solution based on Click. It integrates both DPDK and NetMap in Click and therefore provides I/O batching, computation batching, multi-queue support, and zero-copy to Click. Thanks to these features, FastClick improves Click throughput performances.

B. Netmap

Netmap [10] is a framework which allows commodity hardware (without modifying applications or adding custom

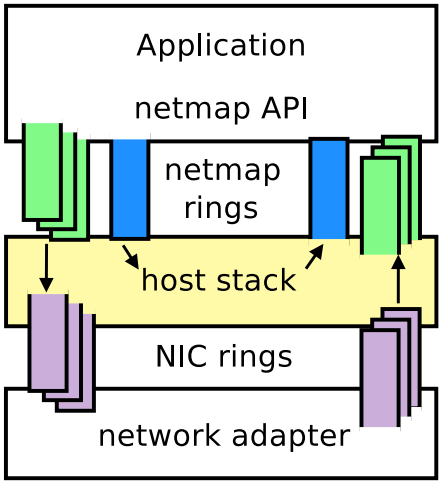

Fig. 3. Netmap [10].

hardware) to handle millions of packets per second which go over 1..10 Gbit/s links. In [10] they claim to remove three main packet processing costs, namely: per-packet dynamic memory allocations (removed by preallocating resources); system call overheads (amortized over large batches) and memory copies (by sharing metadata and buffers between kernel and user-space), while the protection of access to device registers and other kernel memory areas still remains. The main goal of netmap is to build a fast path between the NIC and the applications [54].

The provided architecture is flexible, does not depend on specific hardware and is tightly integrated with existing operating system primitives. The implementation of netmap has been done in Linux and FreeBSD with minimal modifications for several 1 and 10 Gbit/s network adapters. For the minimum Ethernet frame size of 64 bytes, there are also 7 bytes of preamble, 1 byte which represents the start of the frame delimiter and 12 bytes of the interpacket gap. In total, 160 additional bits. This means that on the 10 Gbit/s link, one can achieve the rate of 14.88 Mpps or, in other words, one packet is sent every 67.2 nanoseconds. According to [10], a speed of 14.88 Mpps can be reached with a single core 900 MHz processor which represents the maximum packet rate for the 10 Gbit/s link and the same core can reach well above capacity of 1 Gbit/s link just by running at 150 MHz.

Each network interface can be switched between regular and netmap mode. Regular mode is a standard mode where NIC exchanges packets with the host stack. In netmap mode, NIC rings are disconnected from the host network stack and packets are transferred through the netmap API (Application Programming Interface). An application is allowed to exchange packets with the host stack and with the NIC through *netmap rings* that are implemented in shared memory. This is shown in Figure 3.

Netmap uses different techniques in order to attain its high performance [10]:

- a metadata representation which is easy to use, lightweight, compact, and which hides device-specific features;
- useful hardware features such as multiple hardware queues are supported;
- zero-copy packet transfer between interfaces is supported along with a removal of data-copy costs by giving applications direct and protected access to packet buffers;
- memory preallocation for the linear, fixed size packet buffers when the device is opened in order to save the cost of per-packet allocations/deallocations;

C. NetSlices

NetSlice [55] represents operating system abstraction which processes packets in user-space and enables a linear increase of performance with the number of cores. Memory, multi-queue NIC resources and CPU cores are assigned exclusively (rather than time shared) in order to lower the interconnect and overall memory contention. Marian *et al.* [55] claim to closely match the performance of state-of-the-art in-kernel RouteBricks variants with their user-space packet processors which are built with NetSlice. Unlike alternative user-space implementations which rely on a conventional raw socket, NetSlice operates at nominal 10Gbps network line speeds. The only requirement of NetSlice is a simple kernel extension which can be loaded at runtime, it is not dependent on hardware configuration and provides a replacement for conventional raw sockets.

The main problem with raw sockets is that they were originally made when the ratio between CPU performance and network speed kept being the same over time. When many cores were introduced on the same silicon chip, it changed the paradigm because the number of cores per chip has been increasing while the single core performance has been stationary for years. NetSlice authors claim that:

- Raw socket abstraction provides user-mode application without control of physical resources involved in packet processing.
- Besides being simple and common to all types of sockets, the socket API is not efficient since it needs a system call for every packet send/receive operation.
- There is an increased contention because packet processing hardware and software resources are loosely coupled.
- The path of the packet between a NIC and a user-space raw endpoint is too expensive.

NetSlice spatial partitioning of resources is shown in Figure 4. Spatial partitioning of hardware resources at coarse granularity is done in order to reduce interference/contention. Also, a fine-grained control of hardware resources is provided by NetSlice API. The streamlined path is also brought by NetSlice for packets which move between user-space and NICs. With spatial partitioning, network traffic is divided into "slices" and independent packet processing execution contexts allow parallelism and contention minimization. Each execution context is called a NetSlice. Every particular NetSlice consists of hardware components like NICs and CPUs, and software components like user-mode task and in-kernel network stack which are tightly connected.

Multiple transmit and receive queues are used in parallel by multiple cores because network speeds have continued

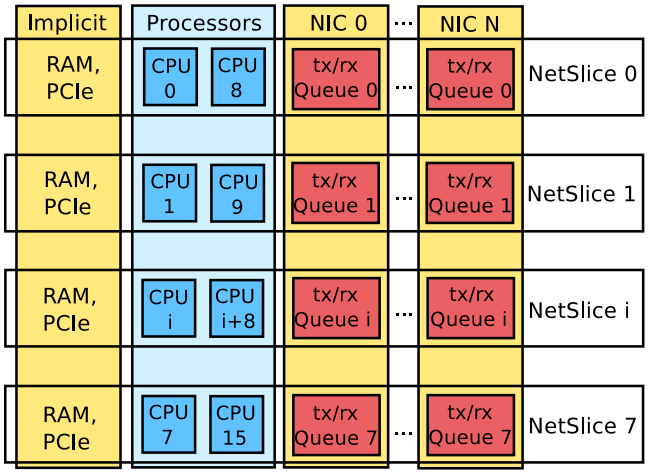

Fig. 4. NetSlice spatial partitioning [55].

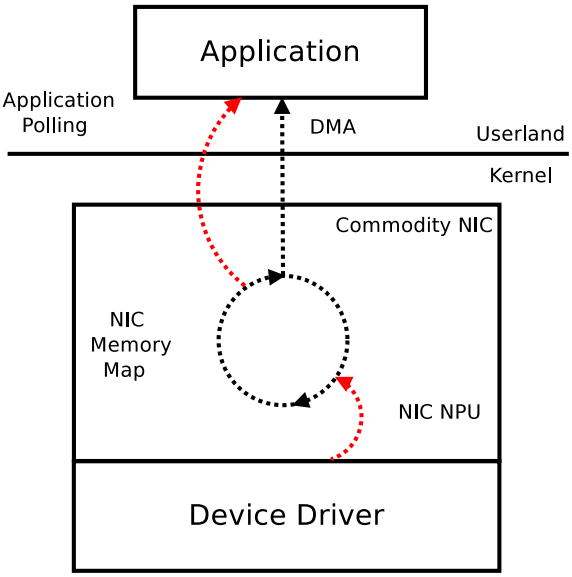

Fig. 5. PF_RING with DNA driver [12].

to increase and CPUs have increased a number of cores rather than the individual core performance. This is why each NetSlice consists of one such tx and one such rx queue per attached NIC and two (or more) tandem CPUs.

Netslice does not use zero-copy techniques but copies each packet once between the user-space and kernel-space. Furthermore, it comprises of a single module which can be loaded at runtime and which works out-of-the-box with vanilla Linux kernels which run stock NIC device drivers.

In the experimental setup from [55], NetSlice has been compared with the default in-kernel routing, implementation of Routebricks and the best configurations pcap user-space solutions. It has been shown that when all CPUs and all NIC queues are used, NetSlice attains nominal line rate of 9.7Gbps for MTU packet size and MAC layer overhead just like the Routebricks and kernel routing. When only a single NIC queue (per available NIC) is used, NetSlice reaches somewhat lower performance, but using more than one NetSlice easily attains the line rate.

D. PF_RING (DNA)

PF_RING [12] is a high-speed packet capture library that allows a commodity PC to perform efficient network measurement which allows both packet and active traffic analysis and manipulation.

The majority of networking tools (*ethereal, tcpdump, snort*) are based on a *libpcap* library which provides a high level interface to packet capture [56]. It allows capturing from different network media, provides the same programming interface for every platform and implements advanced packet filtering capabilities based on Berkeley Packet Filtering (BPF) into the kernel for performance improvement. *Interrupt livelock* [57] limits the possible performance of packet capturing. In fact, whenever a NIC receives a packet, it informs the OS by generating an interrupt, which turns out to take up the majority of CPU time in the case of high traffic rate. This is why *device polling* is introduced. When the OS receives an interrupt from the NIC, it masks future interrupts generated by the NIC (so that NIC can not interrupt the kernel) and schedules periodical device polling to service its needs. When the NIC is served by the driver, NIC's interrupts are enabled once again.

Deri *et al.* [56] claim that the mmap-version of *libpcap* reduces the time needed for moving a packet from kernel to user-space but does not improve the journey from NIC to the kernel. This is why PF_RING socket is introduced. It allocates a memory buffer at socket creation and deallocates it when a socket is deactivated. Incoming packets are copied to the buffer and they are accessible by user-space applications. In this way, system calls are avoided when reading packets.

For reaching maximum packet processing speed PF_RING can use zero-copy drivers (PF_RING ZC) which allow achieving speeds of 1/10Gbit line rate packet processing (both TX and RX) for any packet size. PF_RING consists of:

- *PF_RING Kernel Module* - responsible for low-level packet copying to the PF_RING circular queues;
- *user-Space PF_RING SDK* - provides transparent PF_RING-support to user-space applications;
- *PF_RING-Aware Drivers* - allow additional improvements in packet capturing by efficiently copying packets from the driver to the PF_RING without passing through the kernel.

PF_RING implements a new type of network socket (PF_RING). This allows user-space applications to speak with the PF_RING kernel module. In the case of ZC drivers, packets are read directly from the network interface. Both the Linux kernel and the PF_RING module are bypassed, therefore, the CPU utilization for copying packets to the host is 0%.

Moreover, PF_RING DNA (Direct NIC Access) [48], [50] uses a memory ring allocated by the device driver to host pointers to incoming packets, instead of using a per-socket PF_RING circular buffer. PF_RING DNA is shown in Figure 5. A modified network driver allocates a memory ring using continuous non-swappable memory. The network adapter copies the received packets to the ring and it is up to the user-space application, which manages the buffer, to read

packets and update the index of the next available slot for the incoming packet. PF_RING maps in user-space network card registers and packet memory ring as all the communication with network adapter is done in DMA.

PF_RING polls packets from NICs by using Linux NAPI [12]. NAPI [58] copies packets from the network adapter to the PF_RING circular buffer and the user-space application reads packets from the ring. In PF_RING DNA, NIC memory and registers are mapped in the user-space so that packet copy from the network adapter to DMA ring is done by the NIC NPU (Network Process Unit) instead of NAPI. In this way, CPU is used only for consuming packets and not for transfer from the NIC. PF_RING DNA is shown in Figure 5.

E. DPDK

DPDK [9] is a set of data plane libraries and drivers which are used for fast packet processing. It has provided support for Intel x86 CPUs, which is now extended to IBM Power 8, EZchip TILE-Gx and ARM. DPDK's main goal is to provide a framework for fast packet processing in data plane applications that is simple and complete. This allows a transition from separate architectures for each major workload to a single architecture which unites workloads in a more scalable and simplified solution [59]. The 6WINDGate [60] solution implements an isolated *fast path* networking stack which relies on DPDK in order to accelerate the packet processing speed.

For packet processing, DPDK can use either run to completion or pipeline model. There is no scheduler and all devices are accessed by polling, because otherwise interrupt processing would impose performance overhead. Environment Abstraction Layer (EAL) [9] provides a generic interface and hides environment specifics from the libraries and applications. EAL allows gaining access to low-level resources such as memory space and hardware. Furthermore, an initialization routine decides how to allocate these resources. Major DPDK components are shown in Figure 6.

DPDK runs as a user-space application using the pthread library, which removes the overhead of copying the data between the kernel-space and user-space. Performances are also improved by using cache alignment, core affinity, disabling interrupts, implementing huge pages to reduce translation lookaside buffer (TLB), prefetching and many other concepts [59].

Key software components of DPDK [59]:

- *Memory Pool Manager* - responsible for allocating NUMA-aware pools of objects in memory. In order to improve performance by reducing TLB misses, pools are created in huge-page memory space and rings are used for storing free objects [59].
- *Buffer Manager* - greatly reduces the amount of time needed for allocating and de-allocating buffers.
- *Queue Manager* - implements lockless queues which are safe, instead of spinlocks, and this allows software components to process packets while avoiding redundant wait times.
- *Flow Classification* - combines packet into flows, which enables faster processing and improved throughput, and also provides an efficient mechanism for hash generation
- *Poll Mode Drivers* - drivers for 1GbE and 10GbE Ethernet controllers significantly improve the speed of packet pipeline by working without asynchronous, interrupt-based signaling mechanisms.

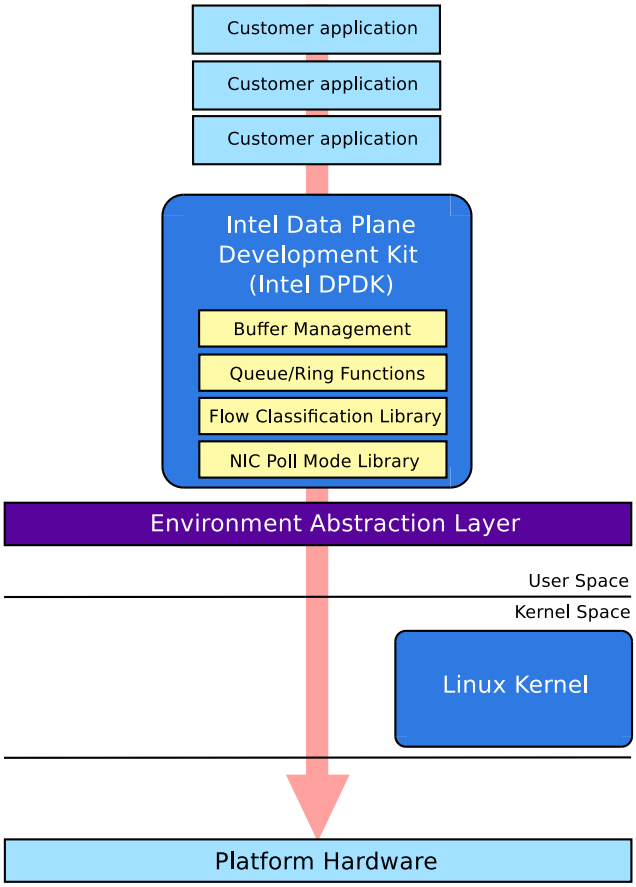

Fig. 6. Major DPDK components [59].

The authors claim that DPDK can improve packet processing throughput by up to ten times and that it is possible to achieve over 80 Mpps throughput on a single Intel Xeon processor (could be doubled with a dual-processor configuration).

IV. Hardware Implementations

In this section, we introduce two types of hardware implementations: the GPU-based solutions and the FPGA-based solutions. We emphasize here that we classified as hardware implementations all the solutions that rely on specific hardware such as GPUs and FPGAs, while the solutions that rely solely on the processing power of CPUs have been described in the previous section. It is for this reason that even multiple solutions that are based on Click Modular Router are represented in this section as they offload part of the processing on a specialized hardware.

A. GPU-Based Solutions

1) Snap: Snap [17] is a packet processing system based on Click which offloads some of the computation load on GPUs.

Its goal is to improve the speed of Click packet processing by offloading heavy computation processes on GPUs while preserving Click's flexibility. Therefore, the authors designed Snap to offload only specific elements to GPUs which, if not offloaded, could have created bottlenecks in the packet processing. The processing performed on the GPU is considered to be of the *fast path* type, while the *slow path* processing is performed on the CPU. To achieve that, Snap adds a new type of "batch" element and a set of adapter elements which allow the construction of pipelines toward GPUs. However, the standard Click pipeline between elements only lets through one packet at a time which is not enough to benefit from GPU offloading. As a matter of fact, offloading only one packet on a GPU results in having a lot of overhead and additionally, a GPU architecture is not adapted to such tasks. Therefore, Snap modifies Click pipeline in order to transmit a batch of packets thus, increasing the performance gain with offloading.

The authors define and implement new methods to exchange data between elements using a new structure called PacketBatch. Thanks to these modifications, Snap provides a GPU-based parallel element which is composed of a GPU part and a CPU part. The GPU part is the GPU kernel code and the CPU part is the part which receives PacketBatches and sends them to the GPU kernel. To improve Snap interaction with the GPU, the authors developed a GPURuntime object programmed and managed with NVIDIA's CUDA toolkit [61]. However, to send and retrieve batches of packets to the GPU, Snap needs two new elements called Batcher and Debatcher. The Batcher collects packets and sends them in batches to the GPU whereas the Debatcher does the inverse, it receives batches of packets and sends them one at a time back to the CPU. Both the Batcher and Debatcher elements manage the necessary copies of data between the host memory and the GPU memory in order to facilitate the use of Snap and also to improve packet copy times.

To benefit from GPU offloading, the GPU has to manage several or all the packets of a batch in parallel. Parallel processing often reorders the packets which is not desirable for TCP or streaming performances. To prevent this issue, Snap uses a GPUCompletionQueue element which waits for all packets of a batch to be processed before sending the batch to the Debatcher.

Snap faces another issue. In Click not all packets follow the same path element-wise, therefore, packets from a batch might have different paths. This separation of packets can happen before reaching the GPU or inside the GPU. The authors found that there are two main classes of packet divergence which are either routing/classification divergence or exception-path divergence. The routing/classification divergence results in having a fragmented memory in the pipeline and happens mostly before reaching the GPU. This is a problem which implies that the unnecessary packets will be copied in the GPU memory which is time-consuming because of the scarce PCIe bandwidth. To solve this issue, Snap only copies the necessary packet in a contiguous memory space at the host level to create a batch of packets which is then sent to the GPU memory using a single transfer through the PCIe. Regarding path divergence happening inside the GPU, Snap solves this issue by attaching predicate bits to each packet. This way the path divergence is only treated once the batch is back in the CPU part thanks to the predicate bits which indicate which element should process the packet next.

In order to further improve Snap performances, the authors used packet slicing to reduce the amount of data that needs to be copied between the host memory and the GPU memory. This slicing mechanism, defined in PacketBatch, allows GPU processing elements to operate on specific regions of data of a packet. For example, to realize an IP route lookup only the destination address is needed, therefore, only this IP address has to be copied to the GPU which reduces by at least 94% the size of data being copied when the packet is 64-bytes long.

Snap, thanks to its improvement over Click, enables fast packet processing with the help of GPU offloading. As a matter of fact, the authors realized a packet forwarder with Snap and its performance was 4 times better than standard Click results. Nevertheless, some Snap elements such as the HostToDeviceMemcpy and the GPUCompletionQueue are GPU-specific which means that Snap is not compatible with all GPUs.

2) PacketShader: PacketShader [11] is a software router framework which uses Graphic Processing Units (GPUs), in order to alleviate the costly computing need from CPUs, for fast packet processing. Actually, this solution allows to take advantage of *fast path* processing of packets by exploiting the GPU's processing power. In comparison to CPU, GPU cores can attain an order of magnitude higher raw computation power and they are very well suited for the data-parallel execution model which is typical for the majority of router applications. PacketShader idea consists of two parts:

- optimization of I/O through the elimination of per-packet memory management overhead and batch packet processing
- offloading of core packet processing tasks to GPUs and the use of massive parallelism for packet processing

In [11] it has been shown that the peak performance of one NVIDIA GTX480 GPU is closely comparable to ten X5550 processors. However, in the case of a small number of packets in GPU parallel processing, CPU shows better performance since GPUs prioritize high-throughput processing of many parallel operations over the execution of a single task with low latency, as CPU does [62].

The Linux network stack uses packet buffers as a basic message unit between network layers. There are two buffers, one used for metadata called skb, and a buffer for packet data. Two main problems exist here. One is the buffer allocation and deallocation which happens tens of millions of times per second in multi-10G networks. The other is metadata size in skb which is long (it contains information required by protocols from different layers) and it is overkill for 64B packets [11].

As a solution for the first problem *huge packet buffer* is introduced in [11]. The idea is to allocate two huge buffers for packet data and metadata, instead of having to allocate buffers for each packet individually. Both buffers contain fixed-size cells and each cell corresponds to one packet in the RX queue.

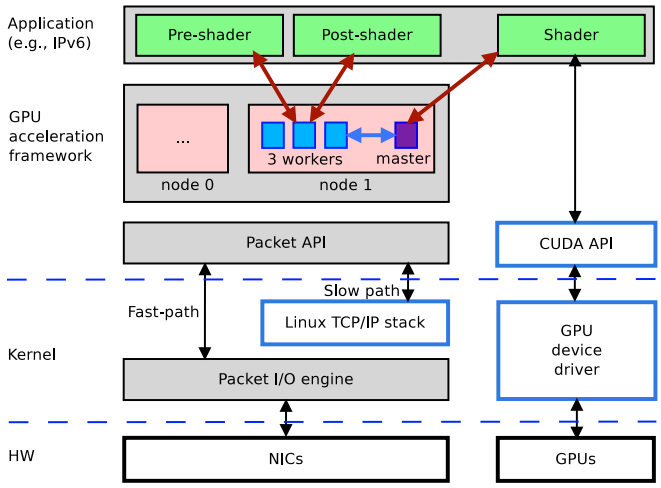

Fig. 7. PacketShader software architecture [11].

In order to reduce per-packet processing overhead, batch processing of multiple packets is introduced. For batch processing, which can be done in hardware, device driver or application, it is shown in [11] that the throughput increases with the number of packets in the batch.

There are two issues which exist in packet I/O on multi-core systems: load balancing between CPU cores and linear performance scalability with additional cores [11]. The solution for these issues is to use multiple receive and transmit queues through Receive Side Scaling (RSS). NIC can send different packets on different RX queues by hashing 5-*tuples* (source and destination IP addresses, their ports and protocol). Then each of the TX or RX queues corresponds to one of the CPU cores which have exclusive access to their appropriate queues.

According to tests performed in [11], node-crossing memory access has 40-50% increased access time and 20-30% lower bandwidth compared to in-node access. For this reason two decisions were taken in NUMA (Non-Uniform Memory Access) system. First, all data structures must be placed in the same node where they are used (huge packet buffers, packet descriptor arrays, metadata buffers and statistics data of NICs are placed in the same node as the receiving NICs of those packets). The second one is to remove node-crossing I/O transactions which are caused by RSS. It turns out that there is 60% performance improvement with NUMA-aware data placement and I/O compared to the NUMA-blind packet I/O.

With all the previously described techniques, with RX and TX together, results show that the minimal forwarding performance is above 40Gbps for packet sizes ranging from 64 to 1514 bytes.

The simplified architecture of PacketShader is shown in Figure 7 (implementation of PacketShader is marked by grey blocks). In order to avoid situations where multiple CPU threads access the same GPU and cause frequent context switching overheads, CPU threads are divided into *worker* and *master* threads. Worker threads request from master to act as a proxy for communication with the GPU and are responsible for packet I/O. A master thread has an exclusive communication with the GPU in the same node for acceleration. Quad-core CPU of each node runs three worker threads and one master thread. Packet processing in PacketShader is divided into three phases [11]:

- *Pre-Shading* - Worker threads get chunks of packets from their own RX queues, classify them for further processing with GPU and drop the malformed ones. Then they build data structures in order to feed input data to the *input queues* of their master threads.
- *Shading* - Input data from host memory is brought by the master thread to GPU memory, then it launches the GPU kernel and transfers back the results from GPU to host memory. After this, the results are placed back to the *output queue* of the worker thread for post-shading.
- *Post-Shading* - Worker thread gets the results from its output queue and modifies, duplicates or drops the packets in the chunk which depends on the processing results. In the end, worker thread splits the packets in the chunk into destination ports for transmission of packets.

3) APUNet: APUNet [33] is an APU-accelerated network packet processing system that exploits the power of integrated GPUs for parallel packet processing while using a CPU for scalable packet I/O. The term APU stands for an Accelerated Processing Unit that is mostly used by AMD to denote their series of 64-bit microprocessors that integrate CPU and GPU on a single die. The authors of APUNet re-examined the claims from the article [63]. The first claim says that for a majority of applications, the benefits of GPU come from fast hardware thread switching which transparently hides memory access latency rather than from the GPU's parallel computation power. For a profound discussion on latency hiding capability on GPUs, readers can refer to the dissertation available in [64]. The second claim says that for many network applications the use of the optimization techniques of group prefetching and software pipelining to CPU code often makes it more resource-efficient when compared to the GPU-accelerated version.

However, the authors of APUNet [33] have found that the GPU's parallel computation power is important for increasing the performance of many compute-intensive algorithms like cryptographic algorithms (for e.x. RSA, SHA-1/SHA-2) that are used in network applications. They have also found a reason for relative performance advantage of CPUs over GPUs. They claim that this is because of the fact that there is a PCIe communication bottleneck between the CPU and a discrete GPU, rather than because of the insufficient capacity of GPUs. For instance, PCIe bandwidth is order of magnitude smaller than the memory bandwidth of GPU. Consequently, they propose the use of integrated GPUs in APU platforms as an economical packet processing solution.

Besides the advantages of the APUs for packet processing applications, there are some drawbacks that need to be addressed. Unlike discrete GPUs, integrated GPUs do not benefit from the high-bandwidth memory of GDDR (Graphics Double Data Rate). Actually the advantage of GDDR when compared to DDR is that it has a much higher bandwidth (with wider memory bus) and it consumes less power. In fact, the integrated GPU needs to share the DRAM with a CPU and they need to contend for the shared memory bus and

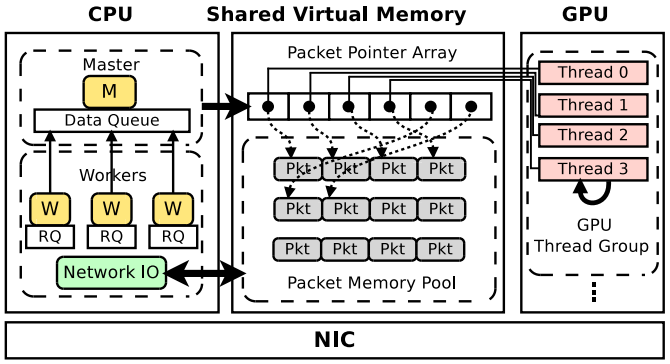

Fig. 8. APUNet architecture: M (Master), W (Worker), RQ (Result Queue) [33].

controller which impacts the efficiency of memory bandwidth usage which can degrade the performance. For this reason, the APUNet solution uses zero-copy technique in all stages of packet processing: packet reception, processing on CPU and GPU and packet transmission [33]. Additionally, in order to obtain low-latency communication between a CPU and a GPU, the APUNet performs persistent GPU kernel execution. In this way, the GPU threads are running in parallel for a continuous input packet stream. Furthermore, a solution to a problem of cache coherency between a CPU and a GPU is proposed through a technique of synchronization of cache memory access by integrated GPU to make the processing results of a GPU available to CPU at low cost [33].

Just like PacketShader [11], the APUNet uses a single-master, multiple worker framework as shown in Figure 8. A dedicated master communicates only with the GPU and the workers perform the packet I/O and request, through the master, the packet processing with the GPU. The worker threads and the master thread are affinitized to one of the CPU cores. Packet memory pool which stores an array of packet pointers and packet payload is shared between a CPU and a GPU. Batch of packets is read by using the DPDK and worker threads process a portion of packets that is load-balanced through the RSS technique. The master notifies a GPU about the available packets which have been previously validated by the worker threads. After having processed the packets, a GPU notifies the master who notifies the worker threads of the results and afterward the packets get transmitted to the appropriate network port.

4) GASPP: GASPP [34] is a programmable network traffic processing framework that was made for modern GPUs. Many of the typical operations used by different types of network traffic processing applications are integrated into this solution as GPU-based implementations. This allows the applications to scale in terms of performance and to process infrequent operations on the CPU.

The problem with the use of GPUs for packet processing applications is that programming abstractions and GPU-based libraries are missing even for the most simple tasks [65]. Moreover, packets need to be transferred from the NICs to the user-space context of the application and after that through the kernel in order to get them to the GPU. For this reason, an increased effort is needed to develop high-performance GPU-based packet processing solutions even for the simpler cases. The authors of GASPP claim that their solution allows to create packet processing solutions flexibly and efficiently. They have developed mechanisms that avoid redundant data transfers by providing memory context sharing between the NICs and a GPU. Additionally, efficient packet scheduling allows better utilization of the GPU and the shared PCIe bus. The main advantages of GASPP are [65]:

- purely GPU-based implementation of a TCP stream reconstruction and flow state management
- packet scheduling technique that tackles load imbalance across GPU threads and control flow irregularities
- zero-copy mechanism between a GPU and the NICs which improves the throughput between the devices

Stateful Protocol Analysis component of GASPP is conceived for maintaining the state of TCP connections and reconstructing the application-level byte stream. This is performed by merging packet payloads and reordering out-of order packets [65]. All the states of TCP connections are saved in the global device memory of the GPU. Despite the fact that batch processing handles out-of-order packets from the same batch, it can not handle the out-of order problem for the packets from different batches. In order to be able to deal with TCP sequence hole scenarios, GASPP processes exclusively the packets that have sequence numbers lower or equal to the connection's current sequence number.

GASPP uses a single buffer for efficient data sharing between a GPU and the network interfaces by adjusting the Netmap module. This allows to avoid costly packet copies and context switches [65].

The authors claim that GASPP achieves multi-gigabit forwarding rates for the computationally intensive network operations like intrusion detection, stateful traffic classification and packet encryption [65].

B. FPGA-Based Solutions

1) ClickNP: ClickNP [20] represents an FPGA-accelerated platform for high performance and highly flexible Network Function (NF) processing on commodity servers. In order to enable multi-tenancy in the cloud and provide security and performance isolation, each tenant is deployed in *virtualized network* environment. Software NFs on servers are used for maximizing the flexibility because conventional hardware-based network appliances are not flexible. Nevertheless, there are two main drawbacks of software NFs. First one is the limited capacity of software packet processing because usually multiple cores are needed to achieve a rate of 10 Gbps. The second one is large and highly variable latency.

In the case of ClickNP, FPGA is used to overcome the limitations of software NFs even though there already exist some proposals which accelerate NFs by using GPUs or Network Processors (NP). Authors claim that FPGA is more power efficient compared to GPU and that it is more versatile than NPs because it can be virtually reconfigured with any hardware logic for any service.

Despite that FPGAs are not expensive, the main challenge that stays behind FPGAs is their *programmability*.

FPGAs are programmed in low-level hardware description languages (HDLs) like Verilog and VHDL which are complex, result in low productivity and are prone to debugging difficulties.

Nevertheless, ClickNP tackles the programming challenges with 3 approaches. First, ClickNP has a modular architecture just like the previously described Click modular router and each complex network function is decoupled and composed of well-defined elements. Second, elements of ClickNP are written with a high-level C-like language and they are cross-platform [20]. ClickNP elements can be compiled into binaries on CPU or low-level HDL, by using high-level synthesis (HLS) tools. The third thing is that high-performance PCIE I/O channel is used between CPU and FPGA. This channel has low latency and high throughput which allows joint processing on CPU and FPGA, by arbitrarily partitioning the processing between the two. The FPGA in this case is used as a *fast path* where the advantage is the increase in the operations processing speed.

The main goal of ClickNP was to build a platform which is flexible, modular, reaches high performance with low latency and supports joint CPU/FPGA packet processing. The authors of ClickNP [20] claim to achieve packet processing throughput of up to 200 million packets per second which is 10 times better throughput compared to state-of-the-art software NFs on CPU and 2.5 times better throughput compared to the case of CPU with GPU acceleration. ClickNP also achieves the ultra-low latency of 2μs which is 10 times and 100 times lower than state-of-the-art software NFs on CPU and the case of CPU with GPU acceleration, respectively. The reason that the GPU in combination with CPU produces higher latency is further discussed in Section VI-B.

2) GRIP: Bellows *et al.* [30] point out that transmitting or receiving data at gigabit speeds already fully monopolize the CPU, therefore, it is not possible to realize any additional processing of these data without degrading throughput. If data were to be processed at either the application layer or the network layer then the CPU should share its computation time between transmitting/receiving data and processing the data which would result in a lower throughput. In order to provide data processing possibilities without degrading throughput, the authors propose an offloading architecture whose goal is to offload a significant amount of the network processing to a network interface. In this way, the host CPU can focus on processing the data. The authors have prototyped the GRIP (Gigabit Rate IPSec) card which alleviates the load of the CPU. This card is based on an FPGA and a commodity Gigabit Ethernet MAC [30]. The card is a SLAAC-1V based on the Xilinx Virtex architecture which provides three user-programmable Virtex FPGAs. In Figure 9 we can see those three FPGA as X0, X1, and X2, we can also see at the top the GRIP card which provides the Gigabit Ethernet I/O extension for the SLAAV-1V. This GRIP card focuses on packet reception and filtering mostly for jumbo frames needed for IPSec. After this filtering, the packet is sent to the X0 chip which plays the role of the bridge between the three other chips. Among those three chips, X1 manages encryption for outbound packets, and X2 manages decryption for incoming

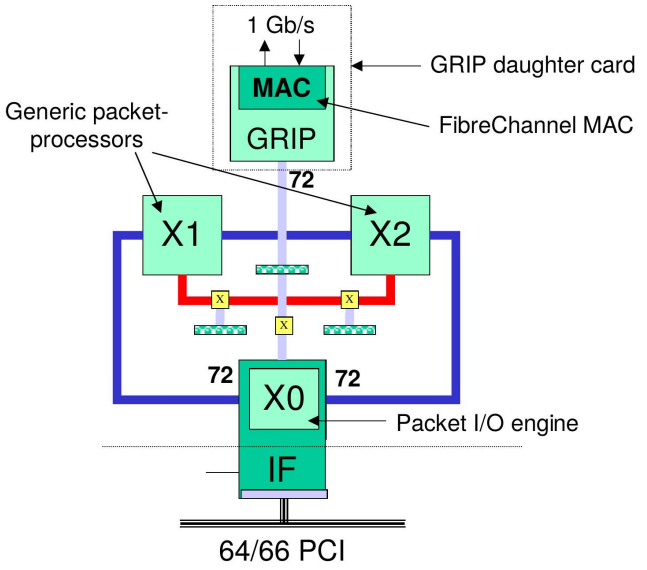

Fig. 9. Block diagram of the SLAAC-1V architecture modified for GRIP [30].

packets. However, the most important part of the GRIP solution in order to achieve fast packet-processing is the Direct Memory Access (DMA) engine located in the X0 chip. The original SLAAC-1V DMA engine has been modified four times. The first modification has been performed in order to work with 64/33 or 64/66 PCI thus, reaching a bi-directional processing of 1Gb/s. The second modification was to add deep scatter-gather tables to allow the process of 255 independent packets between host interrupt responses. The third modification adds a 64-bit 8-way barrel rotator in order to perform transfers with arbitrary byte-alignment. Lastly, adding logic to the DMA engine allows the generation of the framing bits for packet boundaries.

On the software side, GRIP can be divided into two parts. On the one hand there is a generalized driver and on the other hand, there is a modified kernel/application function. The generalized driver role is to make the GRIP card appear as a standard Gigabit Ethernet network interface for the operating system. The modified kernel is only restricted to the FPGAs on the card. It's this part of the software which manages the application-specific offloading, meaning that in the GRIP project this code manages the offloading of all the IPSec protocol layer encryption and decryption. The GRIP solution is a proof of concept showing that using hardware based offloading can improve packet processing speed and therefore overall throughput. However, the GRIP architecture is only focused on offloading the management of the IPSec protocol but the authors will work on improving the GRIP architecture by adding more network function offloading possibilities.

3) SwitchBlade: SwitchBlade [29] represents a platform which is used for rapid prototyping and deployment of custom protocols on programmable hardware. It was developed out of the need for a unique platform which can provide fast prototyping of new protocols but at the same time reach high-performance level which is difficult to attain with purely software router deployments. Although other solutions were

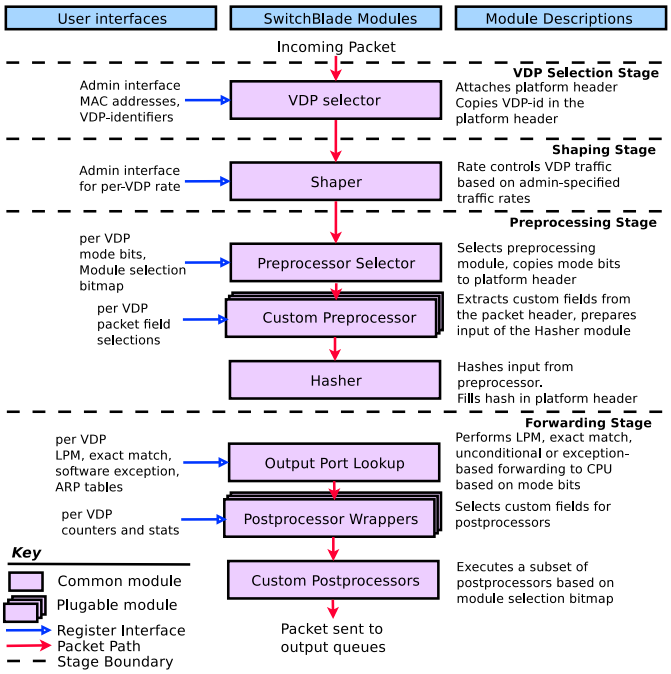

Fig. 10. SwitchBlade Packet Processing Pipeline [29].

targeting the same needs for programmable and fast routers, the authors claim that they fell short for different reasons like portability, scalability and cost of high-speed chips.

Wire-speed performance and the ease of programming, which are provided by SwitchBlade, allow rapid prototyping of custom data-plane functions which can be directly deployed into a production network. SwitchBlade needs to encounter several challenges [29]:

- *Design and Implementation of a Customizable Hardware Pipeline* - SwitchBlade's packet processing pipeline consists of hardware modules that implement common data-plane functions. A subset of these hardware modules can be selected on the fly by new protocols, without resynthesizing hardware [29].
- *Seamless Support for Software Exceptions* - In the case when custom processing elements can not be implemented in hardware, SwitchBlade needs to be able to invoke software routines for processing.
- *Resource Isolation For Simultaneous Data Plane Pipelines* - On the same hardware, multiple protocols can run at the same time in parallel. In this way, each data plane is called VDP (Virtual Data Plane) and each one is provided with separate forwarding table.
- *Hardware Processing of Custom, Non-IP Headers* - modules which process appropriate header fields are provided by SwitchBlade in order to make forwarding decisions.

There are three main design goals sorted by priority [29]:

1) *Rapid Development and Deployment on Fast Hardware* - Design of new protocols often requires changes in the data plane which usually results in a performance drop in the case when the implementation is done entirely in software. In this way, these protocols can not be evaluated with the data rates of production networks. In order to provide wire-speed performance for designers of new protocols, SwitchBlade is implemented using NetFPGA but it can be implemented with any FPGA. NetFPGA's are programmable, they are not tied to specific vendors and provide acceptable speeds.
2) *Customizability and Programmability* - since new protocols share common data plane extensions and in order to shorten the turnaround time for hardware based implementations, SwitchBlade provides a rich set of extensions which are available as modules and allows protocols to dynamically choose the needed subset of modules. Modules of SwitchBlade are programmable and can operate on any offset within packet headers. New modules can be either added in hardware or implemented as exception handlers in software.
3) *Parallel Custom Data Planes on a Common Hardware Platform* - SwitchBlade provides the possibility of running customized data planes in parallel where each data plane is called Virtual Data Plane (VDP). Each VDP has its own virtual interfaces and forwarding tables. Per-VDP rate control is used in order to make isolation between VDPs. SwitchBlade ensures that data planes do not interfere with each other even tough they share hardware modules.

There are several unique design features which allow for rapid development and deployment of new protocols with wire-speed performance.

SwitchBlade pipeline architecture consists of four main stages where each stage contains one or more hardware modules as can be seen on Figure 10. Pipelined architecture is used because it is the most straight forward one and NetFPGA also has a pipelined architecture.

In the already mentioned custom VDPs, each VDP can provide custom modules or share modules with other VDPs. Shared modules are not replicated on the hardware in order to save resources. VDP identifiers are included in software exceptions which makes it easy to separate software handlers which belong to different VDPs.

The behavior of existing modules can be changed in the case when different VDPs need to use the same postprocessing module on different parts of a packet header, for different protocols at the same time. In order to prevent a waste of resources by introducing the same module twice, *wrapper modules* are used. They can customize the behavior of existing modules within same data word and for the same length of data.

Software exceptions are used in order to avoid the implementation of complex forwarding in the case when it is too expensive or when it needs to be performed on a minority of packets whose processing requirements are different from the rest of the packets. Development, in this case, can be faster and less expensive if those packets are processed in the CPU.

4) Chimpp: Chimpp [27] is a development environment for reconfigurable networking hardware that is based on the Click modular router and that targets the NetFPGA platform. Click router's modular approach with a simple configuration language is used for the design of hardware-based packet processing systems. Click router can be used in combination with

Chimpp in order to provide highly-modular, mixed software and hardware design framework [27].

Despite the fact that designing FPGA is easier than designing ASICs, the typical tools that are used for designing FPGAs are not familiar to many network experts and it takes considerably more time to develop the packet processing solutions with FPGAs. The main goal of the Chimpp solution is to provide a development environment that decreases the effort related to integration and extension of packet processing solutions. It uses a high-level Click-like configuration language that allows to build a variety of interesting designs without the use of Verilog [27]. The main contributions of Chimpp are:

- generation of a top-level Verilog file from a high-level script [27]
- the definitions of elements and their interfaces are done with simple XML syntax
- a framework that allows to design systems with Click software elements and Chimpp hardware elements
- network simulator that integrates a combined hardware and software simulation.

The Click part can function either as control plane or as a *slow path* that handles uncommon traffic. Additionally, hardware elements can be converted to software ones in the case when a hardware implementation is unfeasible or the part of processing in question is not critical in terms of performance. Chimpp provides a simple language that allows to instantiate and interconnect its elements at high-level. Scripts that are written in this simple language along with the libraries of existing elements allow to generate a top-level Verilog file for a given design [27]. The NetFPGA card appears to the host's OS as a NIC and therefore the Click router can easily send and receive packets from the NetFPGA as it can use the appropriate interface name. Additionally, software elements can interact with hardware registers by using a known offset from the base address which allows to co-design specific features by using software and hardware elements.

V. Comparison of Software Solutions

In this section, we compare 7 different software implementations of fast packet processing. The solutions included in comparison are: Click Modular router, NetSlices, RouteBricks, Netmap, PF_RING, FastClick and DPDK. They are compared based on the domain in which they operate (user-space or kernel-space), whether they use zero-copy techniques, batch packet processing and parallelism. A summary of comparison results is presented in Table I.

A. Operations in the User-Space and Kernel-Space

The first criterion of our comparison is to determine whether a solution operates in user-space or kernel-space. Click is implemented on general-purpose PC hardware as an extension to the Linux kernel [15]. There are two drivers which can run Click router configurations. One is a kernel-level driver and the other one is a user-level driver which is used mostly for debugging and profiling. It uses BPF in order to communicate with the network. In [66] it has been shown that Click router is three times slower when run in user-space, compared to the case when run in kernel-space.

However, the Netslice [55] solution works in user-space and enables linear performance improvement with the number of cores by using *spatial partitioning* of the CPU cores, memory and multi-queue NICs. This allows to radically reduce overall memory and interconnect contention.

RouteBricks operates in kernel-space since it runs Linux with Click in polling mode. Netmap, just like Netslice operates in user-space.

PF_RING differs from the normal packet capture in a way that applications which rely on libpcap do not need to be recompiled against a modified library (ring/mmap-aware) since received packets are stored in the buffer instead of kernel data structures [56].

FastClick is a high-speed user-space packet processor which is backward compatible with vanilla Click elements [8].

DPDK also runs as a user-space application in order to remove the high overhead of kernel applications and copying of data between kernel-space and user-space.

B. Zero-Copy Technique

Click Modular router does not use a zero-copy technique.

NetSlice does not require modified zero-copy drivers and it does not rely on zero-copy techniques even though it could benefit from them [55]. Netmap uses a simple data model which is well suited for zero-copy packet forwarding. RouteBricks does not use zero-copy techniques. Kernel bypass along with PF_RING ZC (which is a version of PF_RING) also use it, but on the other hand, RAW_SOCKET does not use the zero-copy technique.

FastClick uses zero-copy techniques in both versions, either when using Click with Netmap or DPDK. DPDK benefits from zero-copy technique while ClickNP does not.

C. Batch Processing

Click only handles multiple packets simultaneously at the NIC and device driver level, but processes packets one by one at the application level [11]. However, Kim *et al.* [18] extend the Click modular router by focusing on both I/O and computation batching in order to improve performance by an order of magnitude.

NetSlice also provides efficient batched send/receive operations which reduce the overhead imposed by issuing a system call per operation. For the same reason this technique is used in RouteBricks but beside receiving multiple packets per poll operation (called "poll-driven" batching), "NIC-driven" batching is introduced. NIC driver is extended in order to relay packet descriptors to/from the NIC in batches of a constant number of packets [16]. The authors claim to achieve a performance improvement of 3 times with poll-driven batching whereas an additional improvement of 2 times is attained with NIC-driven batching.

Netmap also uses batching and the author has shown how transmit rate increases with the batch size.

FastClick uses batch processing since both DPDK and Netmap support batch packet transmission and reception.

TABLE I
COMPARISON OF SOFTWARE IMPLEMENTATIONS

	User-/Kernel-space	Zero-copy	Batch packet processing	Parallelism
Click	Implemented as an extension to the Linux kernel. Two drivers: Linux in-kernel and user-level which is used mostly for debugging and profiling	No	None	None (packets are processed individually by each task, while particular parallelism results from the chaining of several tasks)
NetSlices	Runs in user-space while providing a streamlined path for packets between NICs and user-space	No (instead, it copies each packet once between user- and kernel-space while trading off CPU cycles for flexibility and portability)	Yes, by issuing batched packet send and receive operations in order to amortize the overhead produced by per operation system calls	Yes, by providing an array of independent packet processing execution contexts that "slice" the network traffic
RouteBricks	Kernel-space	No	Yes, by using poll-driven batching along with NIC-driven batching	Yes, with parallelism across multiple servers and across multiple cores within a single server
Netmap	User-space	Yes, zero-copy packet forwarding between interfaces requires only swapping the buffer indexes between the Rx slot of incoming and the Tx slot of outgoing interfaces	Yes, while the system call overheads are amortized over large batches	Yes, by spreading the load to multiple CPU cores without lock contention by mapping NIC rings to the equivalent number of netmap rings
PF_RING	In-kernel packet processing by means of loadable kernel plugins. It creates a ring buffer in the kernel and uses a kernel/user memory map to share it with the user-space program	No, but the PF_RING ZC version uses it	None	Yes, through linear system scaling with the number of cores
FastClick	User-space	Yes, in both versions when using Click in combination with Netmap or DPDK	Yes, it is supported since both DPDK and Netmap support it in transmission and reception. BatchElement class is introduced and it provides batch functionality	Yes, as both Netmap and DPDK support it
DPDK	User-space	Yes	Yes, in order to amortize context-switch overhead	Yes, by using per core buffer caches for each buffer pool so that allocation/freeing can be done without using shared variables

PF_RING does not use batch packet processing whereas DPDK uses batch packet processing.

D. Parallelism

RouteBricks is an example of router architecture which parallelizes packet processing tasks across multiple servers and across multiple cores among those servers. It extends the functionality of a Click Modular Router to exploit new server technologies and applies it to build a cluster-based router.

NetSlice provides an array of independent packet processing execution contexts which exploit parallelism by "slicing" the network traffic.

Netmap supports spreading the load to multiple CPU cores without lock contention by mapping NIC rings to the equivalent number of netmap rings. In the case of PF_RING, there is a linear system scaling with the number of cores as each couple {queue, application} is independent of others [48].

FastClick and DPDK also exploit parallelism.

E. Performance Comparison of Different IO Frameworks

In [67] three IO frameworks have been compared: DPDK, PF_RING and netmap. There are four different hardware characteristics which limit the packet processing performance.

- CPU which is considered to be the dominating bottleneck;
- NIC's maximum transfer rate (which is determined by the Ethernet standard);
- version of PCI Express which is used to connect the NIC card with the rest of the system;
- RAM memory which could restrict packet network bandwidth;

The first two characteristics are more important as they are first to limit the packet processing speed. The authors' conclusion is that as long as the processing cost per packet is small enough, the packet processing rate only depends on NIC limitations (set by the Ethernet standard). As processing cost per packet increases, the limitation is no longer imposed by the NIC interface but rather the CPU, which becomes fully loaded and, consequently, limits the final throughput.

Rizzo [10] divided the cost of moving a packet between the application and the network card into per-byte and per-packet costs. The first one relates to a number of CPU cycles necessary to move data to/from the buffer of the network interface. In the case of netmap, this number is zero as these buffers are exposed to the application and there are no per-byte system costs. However in the general case of packet-I/O APIs, a data copy is needed from/to user-space and, accordingly, there is a per-byte CPU cost. The second cost is the dominating one

and the most demanding situation in terms of system load is when the packets are of minimum size (64B).

DPDK compared to PF_RING has a lower CPU cost per packet forwarding operation while PF_RING, in turn, has lower cost than netmap according to [67].

When comparing the influence of batch sizes on the throughput, DPDK turns out to be the best solution which reaches its highest throughput with a batch size of 32 packets. PF_RING almost reaches the same performance with the same number of packets in a batch. On the contrary, netmap reaches its highest throughput with 128 packets in a batch but with lower throughput compared to the former two [67].

The same authors compared three solutions in terms of latency for batch sizes ranging from 8 to 256 packets. In all cases PF_RING produces the smallest latency. DPDK gives a very close result while netmap has significantly higher latency. For a batch size of 16, the latency gradually increases with the batch size for DPDK and PF_RING. For netmap the opposite happens. Nevertheless, for batch size of 256 packets netmap reaches the latency of the other two frameworks.

The I/O model which is implemented in Windows is conceived to be a general I/O platform that supports multiple media types with different characteristics. With the introduction of network virtualization technology it is difficult for general purpose network stacks to keep up with increasing network workload. Consequently, the Network Driver Interface Specification (NDIS) IO model in the Windows Server OS turns out to be insufficient to support new throughput requirements. Furthermore, there are different mechanisms on other OS' which accelerate I/O and make them advantageous when building network intensive applications [68]. For this reason PacketDirect (PD) has been introduced as it boosts the current NDIS model with an accelerated network IO path [68]. The authors claim that it increases the packet per second (pps) count to be an order of magnitude higher by: reducing a number of cycles/packet, reducing latency and by having a linear speed up with use of additional system resources [68]. However, PD wasn't conceived to replace the traditional I/O model but rather to be used in case when it suits the application needs and when there are sufficient hardware resources available.

F. Other Optimization Techniques

There are other techniques which can be used to speed up the packet processing by the use of extended instruction sets such as SSE (Streaming SIMD (Single Instruction Multiple Data) Extensions) and AVX/AVX2 (Advanced Vector Extensions) that allow to improve the parallelism. Intel has expanded the SSE with new versions SSE2, SSE3 and SSE4. The SSE4 is an instruction set that contains 54 instructions where a subset called SSE4.1 contains 47 instructions while a subset called SSE4.2 contains remaining 7 instructions. The AVX2 vector instruction set has 256-bit registers that can process 8 32-bit integers in parallel [63].

In the case when a host, a network and a server chipsets do not support CRC generation in hardware, it is important to have an efficient software-based CRC generation [69]. For this purpose, Intel has introduced a new CRC32 instruction [70].

VI. Comparison and Discussion on the Use of Specific Hardware in Packet Processing Solutions

In Section IV two types of hardware solutions were described: the GPU-based solutions and the FPGA-based solutions. Among the GPU-based solutions we described: Snap, PacketShader, APUNet and GASPP. Among the FPGA-based solutions we described: ClickNP, GRIP, SwitchBlade and Chimpp. All these solutions are compared based on the type of hardware used to offload (a part of) packet processing from the CPU. They are also compared in terms of their usage of CPU (how it cooperates in packet processing) and based on the connection type between the specific hardware and a CPU. Furthermore, the solutions are compared based on the domain in which they operate (user-space or kernel-space), whether they use zero-copy techniques, batch packet processing and parallelism. A summary of comparison results is presented in Table II.

A. Comparison of Hardware Solutions

1) Hardware Used: Among the GPU-based solutions, PacketShader, Snap and GASPP use a discrete GPU, while the APUNet uses the integrated GPU. The main difference between the two is that an integrated GPU shares the DRAM with a CPU as GPU and CPU are manufactured on the same die. Among the FPGA-based solutions, Chimpp and SwitchBlade use the NetFPGA (even though the authors of SwitchBlade claim that it can be implemented with any FPGA), while the ClickNP uses the FPGA and GRIP uses the SLAAC-1V that is based on the FPGA.

2) Usage of CPU: PacketShader uses the CPU by dividing its threads to *worker* and *master* threads. *Worker* threads are used for packet I/O, while the *master* threads are used for communication with a GPU. In Snap, a CPU only needs to perform simple operations while all the operations that need high performance are executed on a GPU. Similarly, ClickNP annotates some of its elements to be executed on a CPU, while the elements that perform better on an FPGA are annotated to be executed on it. In the APUNet solution, even though an integrated GPU shares the DRAM with a CPU, it uses a CPU for scalable packet I/O, while the GPU is used for *fast path* operations. GRIP solution uses the CPU for software exceptions in the case when the processing is less expensive on a CPU or it needs to be done only for a minority of packets. In the same way, SwitchBlade uses a CPU for programmable software exceptions (when development can be faster and less expensive on a CPU) where the individual packets or flows are directed to a CPU for additional processing. GASPP uses a CPU for the infrequently occurring operations while the Chimpp solution uses a CPU for the execution of software elements of the Click modular router.

3) Connection Type: In a PacketShader solution, there are two GPUs that are connected over the PCIe to two IOHs (I/O Hubs) that are further connected to two CPUs. Snap connects CPU to GPU over the PCIe bus. Similarly, ClickNP uses the

TABLE II
COMPARISON OF HARDWARE IMPLEMENTATIONS

	PacketShader	Snap	ClickNP	APUNet
Hardware used	GPU	GPU	FPGA	integrated GPU
Usage of CPU	Yes, by dividing the CPU threads into *worker* and *master* threads. The former ones are used for packet I/O while the latter ones are used for communication with a GPU	Yes, the CPU only needs to perform simple operations and can spend most of its time on packet I/O	Yes, as some of the Click elements are annotated to be executed on a CPU while the others are annotated to be executed on the FPGA	Yes, it uses a CPU for scalable packet I/O by having an integrated GPU manufactured on the same die as CPU while sharing the DRAM with CPU
Connection Type	There are two GPUs which are connected over PCIe to two IOHs (I/O Hubs) that are further connected to two CPUs	CPU-GPU connection is done over the PCIe bus	PCIe I/O channel is designed to provide a high-throughput and low latency connection that enables joint CPU-FPGA processing	The integrated GPU is manufactured on the same die as a CPU
User-/Kernel-space	User-space	User-space	User-space	User-space
Zero-copy	No (huge packet buffer is used instead which is shared between user and kernel)	No	No	Yes, it uses zero-copy in all stages of packet processing: packet reception, processing by CPU and GPU and packet transmission
Batch packet processing	Yes, by batching multiple packets over a single system call to amortize the cost of per-packet system call overhead	Yes, by executing the same element function on a set (batch) of packets in series in order to get a parallel speedup on GPU	Yes, in order to amortize DMA overhead	Yes, it reads the incoming packets in a batch by using the DPDK packet I/O library
Parallelism	Yes, by exploiting massively-parallel processing power of GPU to perform packet processing where each packet is mapped to an independent GPU thread	Yes, by providing elements that act as adapters between serial and parallel parts of processing pipeline in order to exploit the massively-parallel processing power of GPU	Yes, it exploits massive parallelism of FPGA as all element blocks can run in full parallel inside FPGA logic blocks	Yes, it exercises parallel packet processing by running GPU threads in parallel across a continuous input packet stream

	GRIP	SwitchBlade	GASPP	Chimpp
Hardware used	SLAAC-1V (FPGA)	NetFPGA (can be implemented with any FPGA)	GPU	NetFPGA
Usage of CPU	Yes, for software exceptions in the case when the processing is less expensive on a CPU or it needs to be done only for a minority of packets	Yes, for programmable software exceptions (when development can be faster and less expensive on a CPU) where the individual packets or flows are directed to a CPU for additional processing	Yes, a CPU is used for the infrequently occurring operations	Yes, for the execution of software elements of the Click modular router
Connection Type	Connection between a CPU and SLAAC-1V/GRIP card is done over the PCIe bus while the GRIP card is connected on SLAAC-1V card via a 72-pin external I/O connector	Uses the PCI interface to send or receive packets from the host machine	Each CPU is connected with peripherals through a separate I/O hub which is linked to several PCIe slots. A GPU is connected to the I/O hub via a PCIe slot.	Uses the PCIe interface to connect the NetFPGA card with a host machine
User-/Kernel-space	No details	User-space	User-space	No details
Zero-copy	Yes, by moving a majority of packet handling on the network interface, where a stream of received packets can be re-assembled into a data stream on the network interface which facilitates true zero-copy in the TCP/IP socket model	No	Yes, it delivers the incoming packets to user-space by mapping the NIC's DMA packet buffer	No
Batch packet processing	None	None	Yes, by transferring the packets from the NIC to the memory space of the GPU in batches	None
Parallelism	Yes, it can provide hardware implementation of a variety of parallel communication primitives	Yes, by allowing multiple custom data planes to run in parallel on the same physical hardware while the running protocols are completely isolated	Received packets are classified and processed in parallel by a GPU	Yes, as hardware modules all operate simultaneously in parallel

PCIe I/O channel that is designed to provide a high-throughput and low latency connection that enables joint CPU-FPGA processing. For the APUNet solution the integrated GPU is manufactured on the same die as CPU. GRIP is the PCIe bus to connect a CPU and SLAAC-1V/GRIP card, while the GRIP card is connected on SLAAC-1V card via a 72-pin external I/O

connector. Likewise, SwitchBlade uses the PCIe to send and receive packets form the host machine. In the GASPP solution, each CPU is connected with peripherals through a separate I/O hub which is linked to several PCIe slots. A GPU is connected to the I/O hub via a PCIe slot. Finally, Chimpp uses the PCIe interface to connect the NetFPGA card with a host machine.

4) Operations in the User-Space and Kernel-Space: PacketShader operates in user-space in order to take advantage of user-level programming [11]: reliability in terms of fault isolation, integration with third party libraries (e.g., OpenSSL or CUDA) and familiar development environment. The main problem with user-space packet processing is that it can hardly saturate the network infrastructure since it is common today to have 40GbE network interfaces. A typical example would be *raw socket* which is not able to sustain high rates.

Snap represents a Click-based solution which uses a user-space Click router in combination with netmap. Similarly, in [71] the user-space version of the Click Modular router is used where the standard libpcap library has been replaced with the enhanced version running on top of netmap (called netmap-cap). Despite the fact that Click router is faster when run in kernel-space, which we mentioned previously, a speedup of user-space Click router in combination with netmap matches or exceeds the speed of in-kernel Click router [71].

The authors of SwitchBlade claim that their solution offers header files in C++, Perl, and Python. Those header files refer to register address space for the users register interface only [29]. When the register file is included, the programmer can write a user-space program by reading and writing to the regis- ter interface. The register interface can be used to enable or disable modules in the SwitchBlade pipeline.

In the GASPP solution, packets need to be transferred from the NIC to the user-space context of the application and then through the kernel to the GPU. However, GASPP proposes a mechanism which provides sharing memory context between NIC and the GPU.

5) Zero-Copy Technique: PacketShader uses copying instead of zero-copy techniques (e.g., huge packet buffer which is shared between user and kernel) for better abstraction [11]. Copying allows flexible usage of the user buffer like manipulation and split transmission of batched packets to multiple NIC ports. It also simplifies the recycling of huge packet buffer cells.

One of the reasons because of which Snap [17] does not use zero-copy is because of the divergence before reaching the GPU. The other reason is slicing (copying only the part of the packet which is necessary for packet processing, for e.x. destination address).

APUNet uses zero-copy in all stages of packet processing: packet reception, processing by CPU and GPU and packet transmission. GRIP also uses the zero-copy technique by moving a majority of packet handling on the network interface, where a stream of received packets can be re-assembled into a data stream on the network interface which facilitates true zero-copy in the TCP/IP socket model. GASPP solution delivers the incoming packets to user-space by mapping the NICs DMA packet buffer.

6) Batch Processing: PacketShader has a highly optimized packet I/O engine and processes multiple packets in batch. Pipelining and batching are exploited in order to achieve multi-10G packet I/O performance in the software router [11].

The authors of Snap [17] have shown the how the transmit rate increases with the batch size as they extended Click with architectural features which, among the rest, support batched packet processing. Snap adds a new type of "batch" element that can be implemented on the GPU and a set of adapter elements that transfer packets from and to batch elements. ClickNP also uses batching in order to reduce the DMA overhead. APUNet solution reads the incoming packets in a batch by using the DPDK packet I/O library.

Similarly, GASPP solution transfers the packets from the NIC to the memory space of the GPU in batches.

7) Parallelism: PacketShader takes advantage of massively parallel GPU processing power for the purpose of general packet processing. Each packet is mapped to an independent GPU thread. Similarly, Snap also exploits the parallelism of modern GPUs, but is based on Click Modular router unlike PacketShader.

On the other side, ClickNP can run in full parallel by using FPGAs and parallelism can be provided both on element-level, since each element is mapped into a hardware block on FPGA, and inside an element. APUNet exercises parallel packet processing by running GPU threads in parallel across a continuous input packet stream. GRIP solution can provide hardware implementation of a variety of parallel communication primitives. SwitchBlade allows multiple custom data planes to run in parallel on the same physical hardware while the running protocols are completely isolated. Similarly, as other GPU-based solutions, GASPP classifies the received packets and processes them in parallel on a GPU. All of the Chimpp's hardware modules operate in parallel as well.

B. Discussion on GPU-Based Solutions

Programmability and raw performance of GPUs led them to be used in many different applications beyond the most common ones such as graphic presentation, gaming, simulations etc. [72]. GPUs in comparison to CPUs offer extreme thread-level parallelism while CPUs maximize instruction-level parallelism. Packet processing applications are well suited to the data-parallel execution model that GPUs offer. Fundamental architectural differences between CPUs and GPUs are the cause for a faster performance increase of GPUs compared to CPUs. In fact, CPUs are optimized for sequential code and many of the available transistors are assigned to non-computational tasks like caching and branch prediction. On the contrary, GPUs use additional transistors for computation and achieve higher arithmetic intensity with the same transistor count [73].

According to [63], the main strengths of GPU for packet processing compared to CPU are vectorization and memory latency hiding. Vectorization is provided by a large number of processing cores on GPU which makes it suitable for vector processor of packets. In this way, most of the functions such as lookups, hash computation and encryption can be executed

for multiple packets at once. Modern GPUs hide latency using hardware which is more efficient than the mechanisms used by CPU. On the other hand, the weakness of the GPU is the latency which it introduces. Even though GPUs are able to accelerate packet processing, they introduce latency since it takes about 15μs to transfer a single byte to and from a GPU. Moreover, assembling large batches of packets for parallel processing on GPUs introduces additional latency. In order to cope with the latency problem of discrete GPUs, the integrated GPUs can be used instead as they are located on the same die and communicate through the on-chip interconnect instead of PCIe [74] which provides much lower latency. Additionally, integrated GPUs have lower power consumption because of the absence of an I/O hub used for communication between discrete GPU and memory, as stated in [75]. The other GPU weakness is the memory subsystem of the GPU which is optimized for contiguous access compared to random memory access which is required by the networking applications. In this way, GPUs lose an important fraction of memory bandwidth advantage over CPUs [63]. Another disadvantage of GPUs is that their parallel processing can cause serious packet order violations which can affect end-to-end TCP throughput as emphasized in [76]. This happens due to different code complexity which can cause that a latter batch of packets gets processed and returned to the host memory before a former batch of packets. To address this problem Zheng *et al.* [76] proposed a framework called BLOP which preserves the order of batches by lightly tagging the batches in the CPU before transferring them to the GPU and by efficiently reordering the batches in CPU after the GPU processing. There is also a proposal in [77] that offloads indexing of packets to GPUs. The authors claim that with their solution they can achieve an order of magnitude faster indexing throughputs when compared to the state of the art solutions.

The other difference between GPUs and CPUs is that CPUs usually maintain API consistency for many years which is not the case for GPUs. In fact, code which is written for CPUs can be executed on new generations of CPUs much faster than on the previous ones, which is not necessarily the case for GPUs [78]. In order to cope with the problem of backward compatibility, some general purpose programming environments have been introduced. One of the notable examples is CUDA from Nvidia [79] which is a parallel computing platform and programming model which allows sending C, C++ and Fortran code directly to GPU.

There is a proposal called Hermes [80] which represents an integrated CPU/GPU shared memory microarchitecture used for QoS-aware high-speed routing. The memory is shared between CPU and GPU and, thus, the memory copy overhead is removed. Hermes reaches 5X higher throughput when compared to a GPU-accelerated software router [81] with a reduction in average packet delay of 81.2%.

C. Discussion on FPGA-Based Solutions

In a setup used in [82] it has been shown that FPGAs are more energy efficient when compared to GPUs. Mittal and Vetter [83] claim that one can not assert that either of the platforms are the most energy efficient since the evaluation crucially depends on devices and methodology used in experiments. For the majority of works, however, FPGAs are more energy efficient than GPUs, and further, GPUs are more energy efficient than CPUs. Also, FPGAs provide deterministic timing in the order of nanoseconds which is one of the main advantages when compared to CPUs and GPUs.

FPGAs often have a low clock frequency compared to GPUs and CPUs, typically about 200MHz. They also have a smaller memory bandwidth which is usually 2~10GBps, whereas GPUs have 100GBps and Intel XEON CPU has about 40GBps [20]. In contrast, FPGAs have a massive amount of parallelism built-in as they possess millions of LEs (Logic Elements) and thousands of DSP blocks.

On the other hand, FPGA's programming complexity is not negligible which is why a large community of software developers has stayed away from this technology [84]. Nevertheless, ClickNP solution tries to overcome this problem by allowing to program the FPGAs using high-level languages by the use of high-level synthesis (HLS) tools. Similarly, Click2NetFPGA [85] also compiles Click Modular router program directly into FPGA, but authors of ClickNP claim that the performance attained with Click2NetFPGA is much lower compared to their own solution. This is due to several bottlenecks in the design of Click2NetFPGA like packet I/O and memory. Also, unlike ClickNP, Click2NetFPGA does not support joint FPGA/CPU processing.

NetFPGA [86], [87] is an FPGA based platform commonly employed in research environments, which is used for building high-performance networking systems in hardware. A solution like NetFPGA SUME [88], [89] can reach the speed of up to 100Gbps and its low price allows it to be used by the wider academic and research community.

There are also some projects like NetThreads [90] which allow running threaded software on NetFPGA platform by writing programs in C instead of Verilog.

Another highly flexible, programmable, FPGA-accelerated platform is called C-GEP [91]. This platform allows the implementation of different packet processing applications such as monitoring, routing, switching, filtering, classification, etc. The authors claim that this platform is capable of handling 100Gbps Ethernet traffic [91]. However, the information about the packet rate (in Mpps or even Gpps) that C-GEP supports is not provided. Besides the interfaces that support 40Gbps and 100Gbps, this platform can also host multiple 1 and 10Gbps Ethernet interfaces [91]. The C-GEP solution relies on the use of high capacity Virtex 6 FPGA chip.

1) FPGA Routing Algorithms: Routing in FPGAs is concerned with forming connections between the logic blocks distributed inside the FPGA. It makes sure that the signals are transferred from where they are generated to where they are used. The main purpose of an FPGA router is to find a legal route (while supposing that it exists) and to minimize the delay of the critical path in the circuit which can consist of multiple disjoint paths in the routing resource graph (RRG) [92]. The FPGA routing algorithms are needed in order to increase the performance of FPGA-based designs, as the performance of FPGAs is often limited by interconnection delays and not by

the delay of a combinational logic. The routing problem highly depends on the architecture itself and, consequently, the number of routers needed to route FPGAs differs as much as the FPGA architectures in the market [93].

Routing in FPGA is an NP complete problem [94]. It is generally separated in two phases, global routing and detailed routing, by using the divide and conquer paradigm [93]. A global routing represents a macroscopic allocation of wires without focusing on the more microscopic switching between wires [95]. It defines the relative position of routing channels in relation to the positioning of logic blocks, the number of wires in each channel and how each channel is connected to other channels [95]. The detailed routing architecture is concerned with specifying the lengths of wires and the specific switching quantity and patterns between and among wires and logic block pins [95]. There are a number of algorithms that use mixed routing, where the idea is to integrate both phases (global and detailed routing) at the same time in order to ensure accurate estimation of the routing result [93].

A major research breakthrough in FPGA routing has occurred with the invention of the PathFinder algorithm [96]. This is an iterative algorithm that converges to a solution where all signals are routed while the achieved performance is close to the optimal one allowed by the placement. In order to achieve routability, the signals are forced to negotiate for a resource while determining which signal needs a resource the most. The majority of works on FPGA routing are focused on accelerating the PathFinder even though it has been conceived with sequential nature [97]. Recently, a significant amount of research has been performed on accelerating the FPGA routing through parallelism and a detailed survey on parallel FPGA routing techniques can be found in [97]. For further state of the art on FPGA routing algorithms a reader can refer to [93] and [98]–[100].

D. Other Hardware Solutions

Today, one of the most common packet processing hardware offload applications consists of using NPs as accelerators. In the Ericsson whitepaper [101] author claims that NPU (Network Processor Unit) architectures have reached the breaking point as they impose limitations which can not fulfill new requirements. Namely, NPUs processing pipeline has been designed for Layer 3 packet processing and this design can not reassemble packets into flows for Layer 4 processing or Layer 7 inspection. At the same time, there is a small number of software engineers which are capable of programming NPUs. For this reason, networking accelerators for multi-CPU processors have been introduced which close the performance gap to NPUs. High-end multicore SoCs which contain packet-processing hardware blocks attain throughput of 40Gbps and higher. However, when power consumption is compared between the two systems, the estimation is that the latest multicore processors provide 1Gbps/W, while the latest NPUs provide 5Gbps/W according to [101].

Knapp [102] is a packet processing offloading framework for Intel's Xeon Phi co-processor [103]. Xeon Phi co-processors assist the main CPU by adding instruction throughput and high local memory access bandwidth to the system [102]. The authors claim that their preliminary evaluation shows that Xeon Phi attains performance results comparable to CPUs and GPUs and it can reach a line rate of 40Gbps on a single commodity x86 server.

Another recent proposal, which uses specialized hardware called MPPA (Massively Parallel Processor Array), is presented in [104]. The idea of this work is to improve network performance by offloading part of the packet processing of TRILL protocol on a programmable card which integrates 256 processing cores distributed across 16 compute clusters.

VII. Integration Possibilities in Virtualized Environments

In this section, we discuss several integration and usage directions, as well as the constraints of previously described solutions in virtualized environments. Various advantages provided by virtualization technologies like flexibility, isolation, extensibility, resource sharing and cost reduction make them suitable for the area of packet processing despite their inherent drawback of system overhead that they impose.

Among all the described solutions, the majority (8) have their source code publicly available. Those solutions are: Click [105], RouteBricks [106], FastClick [107], Snap [108], Netmap [109], NetSlice [110], PF_RING [111] and DPDK [112]. All these solutions codes are open source and fully available. Whereas, PacketShader code is partially available [113] according to their website [114]. Codes for APUNet, ClickNP, GRIP, GASPP, Chimpp and SwitchBlade are still not available. The main goal of end-users is to be able to efficiently exploit the features of the proposed solutions in virtualized environments, without a considerable waste of achievable performance on traditional proprietary hardware. For this reason the rest of this section is dedicated to various integration possibilities in such environments.

A. Packet Processing in Virtualized Environments

In [115] and [116] Xen hypervisor [117] was used as virtualization layer while the privileged domain Dom0 was used as a driver domain. On top of it, Click Modular Router was used for traffic generation, reception and forwarding. It was shown that forwarding reaches much higher throughput when performed within the driver domain compared to guest domain. The authors found the bottleneck to be a long memory latency which prevents processor from transferring packets fast enough from driver to guest domain in order to follow the increase of data rate. As a solution (called Fast Bridge) they proposed to group packets in containers and process them in group instead of processing them individually. This grouping allowed to reduce the memory latency for each packet. Moreover, even if the memory latency remains unchanged, as we process a group of packets each time instead of only one packet, the processing time per packet is reduced. Consequently, better packet transfer rate is obtained. Furthermore, in [118] this work has been extended to dynamically tune the packet aggregation mechanism by achieving the

best trade-off between throughput and delay given the required flows QoS, number of concurrent VMs and system load.

VALE [36] is a system based on netmap API which implements high performance Virtual Local Ethernet that can be used to interconnect virtual machines by providing access ports to multiple clients (hypervisors or generic host processes). Netmap API is used as a communication mechanism between a host and a hypervisor, and therefore it allows VALE to expose its multiple independent ports to VMM. VALE is implemented as an extension of the netmap module (less than 1000 additional lines of code). However, hypervisor must be extended as well in order to be able to access the new network backend. For this purpose, the authors modified QEMU [119] and KVM [120] hypervisors.

The same authors continued working on different modifications which tend to improve the packet I/O speed in VMs [37]. In the case when only the hypervisor can be modified, implementation of the interrupt moderation mechanism improves the TX and RX packet rates by amortizing the interrupt overhead over multiple processed packets. Interrupt moderation mechanism allows to reduce the number of interrupts by keeping the NIC hardware from generating an interrupt immediately after receiving a packet. Instead, the hardware waits for more packets to arrive before generating an interrupt. For the case when only the guest driver can be modified, introduction of send combining technique can improve the TX rate. This technique can postpone transmission requests until the arrival of the interrupt and, when it happens, all pending packets get flushed (and there is only a single VM exit per batch). Moreover, when both the hypervisor and the driver can be modified, a simple paravirtual extension can overcome the majority of virtualization overheads and reach the highest possible packet rates without a negative impact on latency [37]. Paravirtualization allows to reduce the number of VM exits by establishing a shared memory region through which I/O request exchange between the guest and the VMM can take place without VM exits [37].

The ptnetmap [38] is a Virtual Passthrough solution based on the netmap framework, which is used as the "device" model exported to VMs. Virtual Passthrough is similar to Hardware Passthrough concept where the host OS provides exclusive control of a physical device to the guest OS which is able to run its own OS-bypass or network-bypass code on top of the physical device [38]. The ptnetmap solution allows complete independence from the hardware as it provides communication at memory speed with physical NICs, software switches or high-performance point-to-point links (netmap pipes). The difference between the Virtual Passthrough and the Hardware Passthrough is that the former does not depend on the presence of specific hardware. In fact, this Virtual Passthrough solution focuses on networking and the NIC "device" which is exported to the VM is not a piece of hardware but rather a software port. The code for ptnetmap has been developed as an extension to the netmap framework. An advantage of ptnetmap over Hardware Passthough solution is that the VM is not limited to a specific piece of hardware, so it can be migrated to hosts with different backing devices. Another advantage is that the communication between VMs does not need to go through the PCIe bus which allows running at memory speed and provides very high packet and data rates [38]. Implementation of ptnetmap requires some extensions to netmap code along with minor changes in the guest device driver and the hypervisor. The successor of ptnetmap is ptnet [121] which is a new paravirtualized device model. A main drawback of ptnetmap is that it prevents the use of devices as regular network interfaces for legacy applications running on the guest. This prevents its adoption by upstream communities (Linux, FreeBSD, QEMU, bhyve, etc.) [121]. The ptnet overcomes these disadvantages and the authors claim that its performance is comparable with a widely adopted standard VirtIO [122].

NetVM [39] is a high-speed network packet processing platform built on top of KVM and DPDK library. It provides virtualization to the network by allowing high bandwidth network functions to operate nearly at line speed. In fact, this platform is capable of chaining the network functions in order to provide a flexible, high-performance network element incorporating multiple functions. NetVM facilitates dynamical scaling, deployment and reprogramming of network functions. Moreover, this architecture supports high-speed inter-VM communication, enabling complex network services to be spread across multiple VMs. Its shared memory framework exploits DPDK library in order to provide zero-copy delivery to VMs and between VMs [39]. However, Garzarella *et al.* [38] claim that NetVM has high CPU overhead because DPDK requires active polling to detect events. Despite this fact, NetVM reaches line rate speed of 10Gbps between VMs. The authors of NetVM claim that their solution outperforms the SR-IOV based systems for forwarding functions and also the functions spanning multiple VMs both in terms of throughput and latency [39].

An open source virtual switch with high performance is OvS-DPDK [123] which combines the functionalities of Open vSwitch [40] and DPDK. Open vSwitch is specifically conceived for virtual environments and it generally forwards the packets in kernel-space data path. Fastpath processing consists of packet forwarding according to flow tables which indicate forwarding/action rules. Slowpath is concerned with packets (for e.g., first packet in a flow) which do not match any of the existing entries in the flow table so they are sent for processing to user-space daemon. This allows all the subsequent packets of a flow to be processed in kernel-space as the user-space daemon updates the flow table in a kernel-space. Consequently, the number of costly context switches per packet flow is significantly reduced.

As previously mentioned, DPDK uses Poll Mode Drivers (PMD) for user-space packet processing which allows direct packet transfer between user-space and physical interface while bypassing the kernel network stack. This allows to avoid the interrupt-driven network device models while allowing OvS to take advantage of advanced Intel instruction sets [123]. Hence, in the OvS-DPDK environment, the fastpath is also done in user-space and this boosts performance as there is no traversal of the kernel network stack. In Figure 11, the architecture of OvS-DPDK is depicted. Despite the fact that the OvS-DPDK solution improves the packet processing performance on the general purpose platform considerably, the

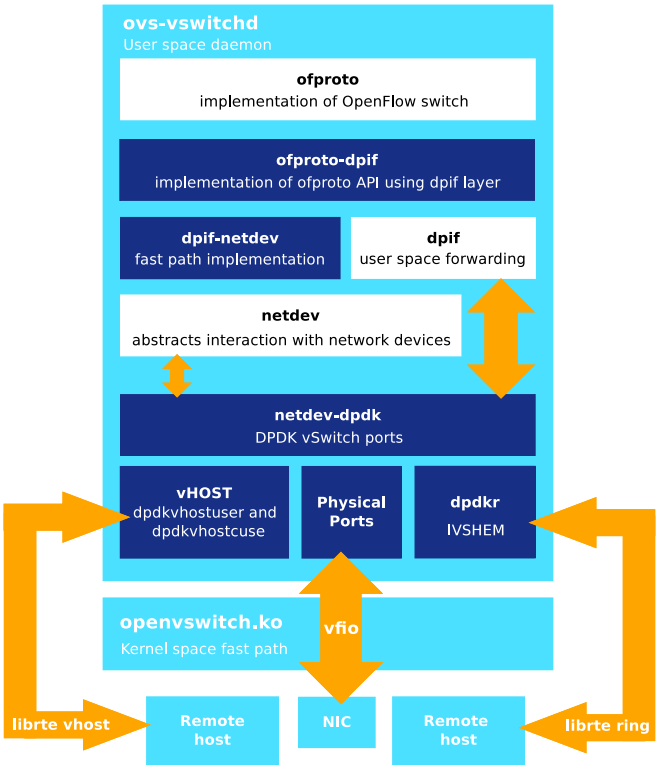

Fig. 11. Open vSwitch with Data Plane Development Kit (DPDK) software architecture [123].

main problem with this solution is that it consumes lots of CPU resources and, consequently, there are less CPU resources left for the applications. Multiple cores are often required just for packet switching and for this reason Tseng *et al.* [124] propose to improve the performance of Open vSwitch by offloading the processing on to the integrated GPUs. The majority of existing approaches are focused on the offloading of packet processing on discrete GPUs. The advantage of integrated GPUs is that they reside on the same die with the CPU offering many advanced features out which the most important ones in terms of packet processing are the on-chip interconnect CPU-GPU communication and a shared physical/virtual memory [74]. The former provides much lower latency while the later allows a GPU and a CPU to share memory space and to have the same virtual address space where there is no more need to copy data back and forth as in the case of discrete GPUs [74]. Tseng *et al.* [124] claim that their OvS-DPDK architecture which uses integrated GPUs improves the throughput by 3x when compared to the CPU-only OvS-DPDK implementation.

Kawashima *et al.* [125] compared how different virtual switches (Linux Bridge, Open vSwitch, VALE, L2FWD-DPDK, OVS-DPDK and Lagopus) that use different packet processing architectures (NAPI, Netmap and DPDK) behave in physical and virtual environments. In their experiments they tested the throughput and latency/jitter of virtual switches installed on a physical server or inside a VM. On a physical server, the L2FWD-DPDK achieved the fastest throughput while OVS-DPDK and VALE reached slightly lower performance. On a VM, the L2FWD-DPDK and OVS-DPDK showed the best performance under the DPDK/vhost-user, but their performance was decreased over 10 Mpps when compared to the physical server experiment. The authors claim that this is due to the vhost-user overhead.

B. Integration Constraints and Usage Requirements

In the previous sections, we presented various software and hardware solutions which aim to increase packet processing speed. Each solution has different software and hardware constraints that mostly depend on the OS version, supported libraries, kernel modules and the underlying hardware which allows the acceleration in packet processing speed. Regarding the ease of use of these solutions, the software solutions seem to have the advantage. As a matter of fact, 7 of those software solutions (Click [105], RouteBricks [106], netmap [109], NetSlice [110], PF_RING [111], DPDK [112], FastClick [107]) do not require a specific hardware. Whereas, the solutions like Snap, PacketShader, GASPP and APUNet require a GPU to process the packets while ClickNP, GRIP, Chimpp and Switchblade use a specific programmable card (based on FPGA). However, even if they do not need a specific network card, software solutions still possess hardware requirements to be fulfilled.

Click requires a specific version of the Linux Kernel with two additional drivers, a user-level driver in order to run the Click application and a Linux kernel driver to run the router configuration. To install this configuration, the user must first write a Click-language configuration file (named config) in the following repository /proc/click/. The original Click system [15] used the Linux's interrupt structure but suffered from the interrupt latency (~50% of the packet processing time) resulting in poor performances. Therefore, to solve this issue, another version of Click [126] uses polling instead of the interruption. However, this implies changes to Linux's structure representing network devices and to the drivers for Click network devices. To use Click it is mandatory to use a modified version of Linux ([126, eq. (2.2.16)]) in order to manage Click's polling device drivers.

RouteBricks is also a software router. As it is based on Click, it has the same constraints as Click. In addition, RouteBricks uses parallelization in order to improve the speed of packet processing by load balancing packets on multiple servers. Therefore, RouteBricks requires multiple servers (cheap and general-purpose ones) to achieve good performances. As a matter of fact, the speed of packet processing depends on the speed of the servers and their number. For example, a 1Tbps RouteBricks router requires 100 servers with a speed of 20Gbps each.

Netmap is the only software solution in this comparison which has been developed for both Linux and FreeBSD. This solution is hardware independent as long as the hardware uses an unmodified libpcap client.

The NetSlice API extends the device-file interface and leverages the flexibility of the ioctl mechanism. It is possible for the user to define user-mode librairies in order to improve or facilitate its use. However, this is not mandatory as

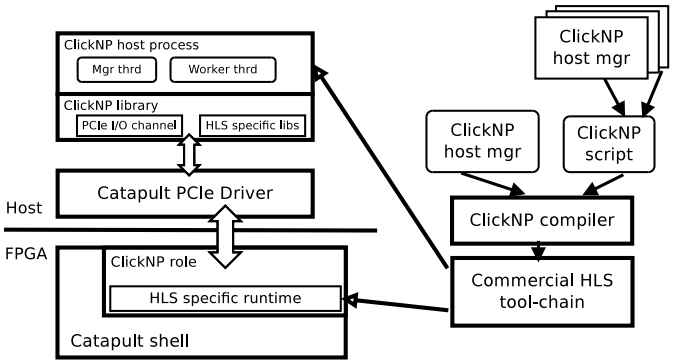

Fig. 12. ClickNP architecture [20].

conventional file operation (read/write/poll) have been mapped to the corresponding NetSlice operations on data flows.

PF_RING is available for Linux kernels 2.6.32 and newer as a module which only needs to be loaded. To poll the packets from the NICs, PF_RING uses Linux NAPI [58] which is an interface to use interrupt mitigation techniques for networking devices in the Linux kernel. However, it requires independent device drivers and as of today only supports cetrain network adapters.

The DPDK solution needs kernel configuration before being used. It requires Linux Kernel 2.6.33 (or newer) and the activation of several options such as:

- UIO support
- HUGETLBFS (Hugepage support)
- PROC_PAGE_MONITOR support
- HPET and HPET_MMAP.

However, in a virtualized environment like Xen, DOM0 does not support hugepages (HUGETLBFS) therefore a new kernel module rte_dom0_mm is needed to allow the allocation (via IOCTL) and mapping (via MMAP) of memory. FastClick requires both netmap and DPDK, and therefore it possesses the same requirements as both solutions. However, in order to integrate both solutions, FastClick proposes a new user-space I/O in Click which implies a new implementation for both Netmap and DPDK.

Snap is a packet processing framework which exploits parallelism available on modern GPUs. The authors used a Linux kernel version 3.2.16. They modified about 2000 lines of Click code, added more than 4000 lines of code for new elements and added more than 3000 lines of code for interaction with GPU. NVIDIA TESLA C2070 was used as a GPU in the experiments. The also used two NICs Intel 82599EB dual-port 10Gbps cards.

The PacketShader solution offloads packet processing on GPUs. However, only NVIDIA GPU can be used as the modification of the GPU driver have been done on the CUDA SDK 3.0 [130]. In more details, the authors used a NVIDIA GTX 480 graphic card on a X16 PCIe link. Additionally, PacketShader requires an unmodified 64-bit Ubuntu Linux 9.04 server distribution (Linux Kernel 2.6.28.10) and its packet I/O engine is based on ixgbe 2.0.38.2 device driver for Intel 10 GbE PCIe adapters [131].

APUNet solution employs an integrated GPU in recent APU platform with a goal to offload a part of packet processing from the CPU to the GPU. The APUNet has been tested on the AMD Carrizo APU platform that represented the packet processing server. On the other hand, a client machine that has an octa-core Intel Xeon E3-1285 v4 (3.50 GHz) with 32GB of RAM has been used for packet generation. A communication is provided via a dual-port 40 Gbps Mellanox ConnectX-4 NICs (MCX414A-B) that use PCIev3 interface. Both machines run an Ubuntu 14.04 (kernel 3.19.0-25-generic).

GASPP is a flexible, efficient and high-performance framework for network traffic processing that exploits the massively parallel processing power of GPUs [34]. GASPP is programmable in C/CUDA language. It uses a single buffer for efficient data sharing between the GPU and the NIC by adjusting the netmap module. The authors used two Intel Xeon E5520 Quad-core CPUs at 2.27GHz and 12 GB of RAM (6 GB per NUMA domain). Each CPU is connected via separate I/O hub to an NVIDIA GTX480 GPU via a PCIe v2.0 x16 slot and to one Intel 82599EB with two 10 GbE ports.

There are few cases for which GASPP is not very well suited or has a limited functionality. For instance, divergent workloads that perform lightweight processing (such as the forwarding application), or workloads for which it is difficult to know in advance execution paths for different packets, may not have an efficient parallel execution on top of GASPP. Additionally, it has a relatively high packet processing latency.

ClickNP seems to be the solution with the most requirements. ClickNP is built on the Catapult Shell architecture [132] and uses its logic in order to communicate with several elements of the host like the PCIe, the DRAM Memory Manage Unit (DMA), the Ethernet MAC. However, ClickNP FPGA program is expressed as a Catapult role and requires commodity High-Level Synthesis (HLS) tool-chains to generate the FPGA Hardware Description Language (HDL). Li *et al.* [20] give a list of three HLS tool-chains compatible with ClickNP which are: 1) Altera SDK for OpenCL [127], 2) SDAccel Development Environment [128], and 3) Vivado Design Suite [129]. Additionally, as those HLS do not produce the same results, the authors provide an HLS-specific runtime which performs the translation between the HLS and the shell interfaces. In Figure 12 we can see that ClickNP requires a ClickNP library for both the PCIe and the HLS communications in the host. Still, in the host, the Catapult PCIe Driver is mandatory to communicate with the FPGA which is considered as a Catapult shell.

GRIP requires the use of its SLAAC-1V high-performance platform, the GRIP card, which presents itself as a standard Gigabit Ethernet network interface to the operating system. In order to offload traffic on the card, a generalized driver is needed to communicate with the card and it is either up to the application or the kernel to offload the traffic. The choice to make is left for the user and therefore the implementation itself. The GRIP driver is based on the SysKonnect [133] Gigabit Ethernet card driver, which also uses the XMACII [134] Media Access Controller chipset.

SwitchBlade, another hardware solution, is based on the NetFPGA reference implementation and on hardware programming in Verilog. It is also possible to communicate with the NIC via SwitchBlade preprocessor. In this case, the

TABLE III
SUMMARY OF THE CONSTRAINTS

Family	Packet processing solutions	Software constraints	Hardware constraints
Software solution	Click	- Linux (Kernel 2.2 +) - User-level driver - Linux kernel driver (support polling)	
	RouteBricks	- Click/Linux - Modified Valiant load balancing (VLB) algorithm	- Multiple general-purpose, off-the-shelf server hardware
	Netmap	- FreeBSD or Linux	- Support Libpcap
	NetSlice	- Linux - File Operation API - ioctl mechanism	
	PF_RING	- Linux (Kernel 2.6.32 +) - Libpcap support - Linux NAPI	- Myricom, Intel and Napatech network adapters - NIC must be libpcap ready
	DPDK	- Linux (Kernel 2.6.33 +) - glibc >= 2.7 - Kernel modules: UIO, HUGETLBFS, PROC_PAGE_MONITOR, HPET, and HPET_MMAP - If Xen then kernel module rte_dom0_mm	- Intel x86 CPUs, IBM Power 8, EZchip TILE-Gx and ARM architectures
	FastClick	- netmap - DPDK	
GPU offloading	Snap	- Linux (Kernel 3.2.16 +)	- Intel 10 GbE PCIe adapters - NVIDIA GPU (CUDA SDK)
	PacketShader	- Linux (Kernel 2.6.28.10 +)	- Intel 10 GbE PCIe adapters - NVIDIA GPU (CUDA SDK 3.0).
	APUNet	- Linux (Kernel 3.19.0-25 +)	- AMD Carrizo APU platform - dual-port 40 Gbps Mellanox ConnectX-4 NIC
	GASPP	- Linux (Kernel 3.5 +) - CUDA v5.0 - netmap	- NVIDIA GTX480 - Intel 82599EB with two 10GbE ports
FPGA offloading	ClickNP	- Windows Server 2012 R2 - ClickNP compiler - 3 Commercial HLS tools ([127],[128],[129]) - ClickNP library - Catapult PCIe Driver	- Catapult Shell architecture - FPGA card supported by HLS tools
	GRIP	- Linux - GRIP driver - Modification of Kernel or application	- GRIP card - PCI 32/33 or 64/66
	SwitchBlade	- Linux - OpenFlow, Path Splicing or IPv6 - OpenVZ container	- NIC must be NetFPGA compliant
	Chimpp	- Linux - OMNeT++ (for simulations)	- NetFPGA card

developer can code in C++, Perl, or Python the new functionalities for the NIC. However, by default, SwitchBlade is restrained to the three following routing protocols and forwarding mechanisms: OpenFlow, Path Splicing, and IPv6. In order to use SwitchBlade in a virtualized environment, the authors used an OpenVZ container because of its two key advantages. OpenVZ provides namespace isolation between virtual environment and provides a CPU scheduler which manages the CPU and usage of memory resources.

Chimpp is a development environment for reconfigurable networking hardware. It uses the NetFPGA card for offloading part of the packet processing from the CPU. It has the same requirements as Click router as it uses it for the software modules, while the hardware modules are executed on NetFPGA card. Otherwise, for the simulation part it uses the OMNeT++ software.

In conclusion, almost all of the solutions require a Linux Kernel either patched or with specific modules activated, only ClickNP uses a Windows kernel. The hardware solutions also require the use of specific cards. It seems that the solution with the least requirements is the netmap solution which only requires an unmodified libpcap client and a NIC which is Libpcap compatible. Table III summarizes all the constraints for all the solutions.

VIII. LATEST APPROACHES AND FUTURE DIRECTIONS

A novel Linux Foundation [136] project called FD.io (Fast data - Input/Output) represents a collection that consists of multiple subprojects which enable Data plane services that are: open source, highly performant, interoperable, multi-vendor, modular and extensible [137], [138]. They aim to fulfill the functional needs of operators, developers and deployers.

One of the main subprojects is concerned with the development of the Vector Packet Processor (VPP) platform [139]. The VPP platform is a framework which can be extended and provides out-of-the-box production quality switch/router functionality [140]. It is also highly optimized packet processing stack for a general-purpose CPU and it leverages the functionality of DPDK. VPP runs as a Linux user-space application.

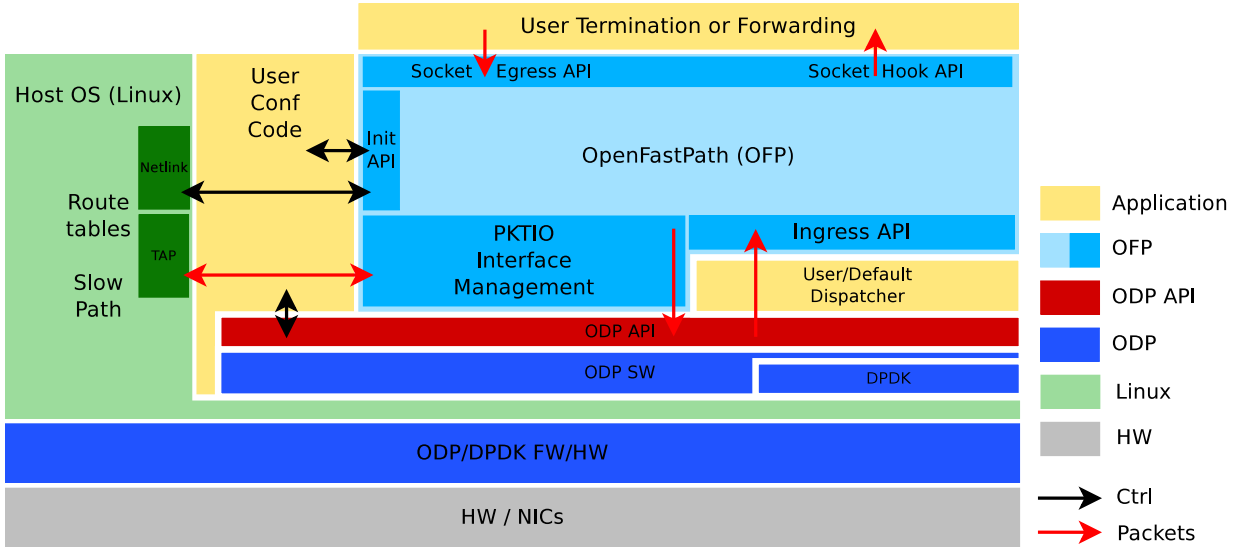

Fig. 13. OpenFastPath system view [135].

It is conceived as a "packet processing graph" to which new customized graph nodes can be easily plugged. One of the advantages of VPP is that it can be run on different architectures like x86, ARM, PowerPC and it can be deployed in VM, container or bare metal environments.

VPP reaches a speed of multiple Mpps per single x86_64 core. The name vector processing refers to the processing of multiple packets at a time, compared to scalar packet processing where only one packet at a time is processed.

One of the latest approaches lead to the integration of VPP on OpenDataPlane (ODP) enabled SmartNICs [141]. ODP [142] is a set of APIs for the networking software defined data plane which are open-source and cross-platform. ODP consists of three parts [143]:

- an abstract API specification that describes a functional model for data plane applications. This allows receiving and transmitting packet data, different types of packet manipulations without specifying precisely how to perform those functions.
- multiple implementations of API specification that are conceived for specific target platforms. Each ODP implementation specifies how ODP API is realized and how abstract types are represented. In this way, ODP API may use specialized instructions to accelerate some API functions or parts of the API may be offloaded to some hardware co-processing engine while saving CPU resources.
- an ODP validation Test Suite which verifies the consistency between different ODP implementations as they must provide the specified functional behavior of ODP API. This is an open source component as well and can be used by application writers, system integrators and platform providers.

Three primary goals of ODP are to allow making portable applications across different platforms, to permit data plane applications to benefit from platform-specific features without specialized programming and to allow applications to scale up automatically to support many core architectures [143]. The main difference between ODP and DPDK is that ODP is API, while DPDK is a specific implementation of an API. ODP is architecturally positioned high enough in order to allow platform abstraction without imposing overheads and strict models. There is also a version of ODP implementation which is integrated with DPDK [144].

Another project intended for packet processing at high-speed is OpenFastPath (OFP) [145]. This project provides an open-source implementation of high-performance TCP/IP stack. The goal of OFP is to allow accelerated routing/forwarding of IPv4 and IPv6 as well as tunneling and termination for a variety of protocols [135]. OFP system view is shown in Figure 13. All functionality which is not supported by OFP goes over the slow path networking stack of the host OS through a TAP interface. OFP boosts throughput and scalability by reducing Linux overhead and provides application portability through the support of ODP and DPDK. OFP functionality is offered as a library to Fast Path applications which are based on ODP run to completion execution model and framework [135]. DPDK is supported as well through the ODP-DPDK layer.

P4 [146] is a high-level language used to program protocol-independent packet processors. This programming language is also target-independent since the packet processing functionality specifications should be provided independently of the underlying hardware. Furthermore, P4 provides reconfigurability as a controller should be able to redefine packet parsing and processing in the field [146]. In this way, a switch is not tied to a specific packet format and a controller can be able to specify a packet parser which extracts particular header fields, and a collection of match+action tables which process these headers. Consequently, P4 provides a programmable parser to allow new headers to be defined, unlike OpenFlow that assumes fixed parser. As the P4 language has received lots of attention in the community, as a result,

multiple solutions intended for different platforms are based on P4 like P4FPGA [147], P4GPU [148], P4-to-VHDL [149] and DC.p4 [150]. Additionally, there is a prototype of the P4 compiler [151] that generates C code directly from a P4 program that uses the Intel DPDK. A possible optimizer for the P4C compiler is presented in [152]. The authors of this work claim that their optimizer provides significant performance improvements.

OpenState [153], on the other hand, is an approach which allows performing stateful control functionalities directly inside a switch without the need for the intervention of the external controller [154]. This solution allows to introduce the states inside a device itself. The abstraction is based on *eXtended Finite State Machines*. OpenState can be considered as an OpenFlow extension and allows an SDN switch to auto adapt itself and update its forwarding behavior without necessitating an interaction with the SDN controller.

Another recent project called BESS (Berkeley Extensible Software Switch) [155] is concerned with building a programmable platform called SoftNIC [156] that augments hardware NICs with software. This platform provides a performance that is comparable to hardware and at the same time the flexibility of software. Han *et al.* [156] claim that their software-augmented NIC solution serves as a fallback or an alternative to hardware NIC. SoftNIC allows developers to build features in software that incure minimal performance overhead. In this way, SoftNIC can be used as a hardware abstraction layer (HAL) to develop software that uses NIC features without thinking about cases when they are not fully available [156].

IX. Conclusion

In this paper, we investigated different types of packet processing on server-class network hosts which try to improve the processing speed by using software, hardware, or their hybrid combination. The main problem about today's networks is that the throughput of the network interfaces, which are in the range of 40Gbps and higher, can be barely supported by the operating system's network stack as it was built for general purpose communication. For this reason, in order to provide high-speed networking applications, a lot of research has been conducted lately which produced various software-based and hardware-based solutions.

We examined multiple solutions based on Click modular router, among which several ones offload Click functions to different types of hardware such as GPUs, FPGAs and parallel cores on different servers. Furthermore, we surveyed some software solutions which are not based on Click modular router and we compared them in terms of a domain in which they operate (user-space or kernel-space), whether or not they use zero-copy techniques, batch packet processing and parallelism. We discussed some hardware solutions as well and compared them additionally in terms of the type of hardware that they use, their usage of CPU and how they connect with it. Moreover, we discussed the integration possibilities in virtualized environments of the described solutions, their constraints and their requirements.

All the analyzed solutions tend to alleviate the same limitations due to the overheads imposed by the architecture of the network stack and their suitability mainly depends on the trade-offs adequacy and the cost required for their implementation.

References

[1] C. V. Networking, "Cisco global cloud index: Forecast and methodology 2015–2020," San Jose, CA, USA, Cisco, White Paper, 2016.

[2] W. Wu, *Packet Forwarding Technologies*. Boca Raton, FL, USA: CRC Press, 2007. [Online]. Available: https://books.google.fr/books?id=ergnzAsxQUoC

[3] *Intel® Ethernet Switch FM10000*. Accessed: Apr. 30, 2018. [Online]. Available: https://goo.gl/eCdzqu

[4] *Broadcom's Tomahawk® 3 Ethernet Switch Chip Delivers 12.8 Tb/s of Speed in a Single 16 nm Device*. Accessed: Apr. 30, 2018. [Online]. Available: https://www.broadcom.com/blog/broadcom-s-tomahawk-3-ethernet-switch-chip-delivers-12-8-tbps-of-speed-in-a-single-16-nm-device

[5] *The Future Is 40 Gigabit Ethernet*, Cisco, San Jose, CA, USA, 2016. [Online]. Available: http://www.cisco.com/c/dam/en/us/products/collateral/switches/catalyst-6500-series-switches/white-paper-c11-737238.pdf

[6] I. Marinos, R. N. M. Watson, and M. Handley, "Network stack specialization for performance," in *Proc. ACM Conf. SIGCOMM*, Chicago, IL, USA, 2014, pp. 175–186. [Online]. Available: http://doi.acm.org/10.1145/2619239.2626311

[7] L. Rizzo, "Revisiting network I/O APIs: The netmap framework," *Queue*, vol. 10, no. 1, pp. 30–39, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2090147.2103536

[8] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *Proc. 11th ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, Oakland, CA, USA, 2015, pp. 5–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=2772722.2772727

[9] Intel. *DPDK: Data Plane Development Kit*. Accessed: Apr. 30, 2018. [Online]. Available: http://dpdk.org/

[10] L. Rizzo, "Netmap: A novel framework for fast packet I/O," in *Proc. USENIX Conf. Annu. Tech. Conf. (USENIX ATC)*, 2012, p. 9. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342821.2342830

[11] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-accelerated software router," in *Proc. ACM SIGCOMM Conf. (SIGCOMM)*, 2010, pp. 195–206. [Online]. Available: http://doi.acm.org/10.1145/1851182.1851207

[12] NTOP. *PF_RING*. Accessed: Apr. 30, 2018. [Online]. Available: http://www.ntop.org/products/packet-capture/pf_ring/

[13] Solarflare. *OpenOnload*. Accessed: Apr. 30, 2018. [Online]. Available: http://www.openonload.org/

[14] *PACKET_MMAP. Linux Kernel Contributors*. Accessed: Apr. 30, 2018. [Online]. Available: https://goo.gl/2YqJyK

[15] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000. [Online]. Available: http://doi.acm.org/10.1145/354871.354874

[16] M. Dobrescu *et al.*, "Routebricks: Exploiting parallelism to scale software routers," in *Proc. ACM SIGOPS 22nd Symp. Oper. Syst. Principles (SOSP)*, 2009, pp. 15–28. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629578

[17] W. Sun and R. Ricci, "Fast and flexible: Parallel packet processing with GPUs and click," in *Proc. 9th ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, San Jose, CA, USA, 2013, pp. 25–36. [Online]. Available: http://dl.acm.org/citation.cfm?id=2537857.2537861

[18] J. Kim, S. Huh, K. Jang, K. Park, and S. B. Moon, "The power of batching in the click modular router," in *Proc. ACM Asia-Pac. Workshop Syst. (APSys)*, Jul. 2012, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/2349896.2349910

[19] B. Chen and R. Morris, "Flexible control of parallelism in a multiprocessor PC router," in *Proc. USENIX Annu. Tech. Conf. Gen. Track*, 2001, pp. 333–346.

[20] B. Li *et al.*, "ClickNP: Highly flexible and high-performance network processing with reconfigurable hardware," in *Proc. ACM SIGCOMM*, Jul. 2016, pp. 1–14. [Online]. Available: https://www.microsoft.com/en-us/research/publication/clicknp-highly-flexible-high-performance-network-processing-reconfigurable-hardware/

[21] N. Shah, W. Plishker, K. Ravindran, and K. Keutzer, "NP-Click: A productive software development approach for network processors," *IEEE Micro*, vol. 24, no. 5, pp. 45–54, Sep. 2004.

[22] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355746

[23] J. C. Mogul *et al.*, "API design challenges for open router platforms on proprietary hardware," in *Proc. HotNets VII*, 2008, pp. 7–12.

[24] O. E. Ferkouss *et al.*, "A 100Gig network processor platform for openflow," in *Proc. 7th Int. Conf. Netw. Service Manag.*, Paris, France, Oct. 2011, pp. 1–4.

[25] Y.-K. Chang and F.-C. Kuo, "Towards optimized packet processing for multithreaded network processor," in *Proc. Int. Conf. High Perform. Switch. Routing*, Richardson, TX, USA, Jun. 2010, pp. 127–132.

[26] T. Wolf and M. A. Franklin, "Performance models for network processor design," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 6, pp. 548–561, Jun. 2006.

[27] E. Rubow, R. McGeer, J. C. Mogul, and A. Vahdat, "Chimpp: A click-based programming and simulation environment for reconfigurable networking hardware," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, La Jolla, CA, USA, Oct. 2010, p. 36. [Online]. Available: http://doi.acm.org/10.1145/1872007.1872052

[28] J. Naous, G. Gibb, S. Bolouki, and N. McKeown, "NetFPGA: Reusable router architecture for experimental research," in *Proc. ACM Workshop Program. Routers Extensible Services Tomorrow (PRESTO)*, Seattle, WA, USA, 2008, pp. 1–7. [Online]. Available: http://doi.acm.org/10.1145/1397718.1397720

[29] M. B. Anwer, M. Motiwala, M. B. Tariq, and N. Feamster, "SwitchBlade: A platform for rapid deployment of network protocols on programmable hardware," in *Proc. ACM SIGCOMM Conf. (SIGCOMM)*, 2010, pp. 183–194. [Online]. Available: http://doi.acm.org/10.1145/1851182.1851206

[30] P. Bellows, J. Flidr, T. Lehman, B. Schott, and K. D. Underwood, "GRIP: A reconfigurable architecture for host-based gigabit-rate packet processing," in *Proc. 10th Annu. IEEE Symp. Field Program. Custom Comput. Mach.*, 2002, pp. 121–130.

[31] M. Attig and G. Brebner, "400 Gb/s programmable packet parsing on a single FPGA," in *Proc. 7th ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, Oct. 2011, pp. 12–23.

[32] A. Bitar, M. S. Abdelfattah, and V. Betz, "Bringing programmability to the data plane: Packet processing with a NoC-enhanced FPGA," in *Proc. Int. Conf. Field Program. Technol. (FPT)*, Dec. 2015, pp. 24–31.

[33] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang, and K. Park, "APUNet: Revitalizing GPU as packet processing accelerator," in *Proc. 14th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Boston, MA, USA, 2017, pp. 83–96. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/go

[34] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis, "Design and implementation of a stateful network packet processing framework for GPUs," *IEEE/ACM Trans. Netw.*, vol. 25, no. 1, pp. 610–623, Feb. 2017, doi: 10.1109/TNET.2016.2597163.

[35] Berten. *White Paper—GPU vs FPGA Performance Comparison*. Accessed: Apr. 30, 2018. [Online]. Available: https://goo.gl/raKHdS

[36] L. Rizzo and G. Lettieri, "VALE, a switched Ethernet for virtual machines," in *Proc. ACM 8th Int. Conf. Emerg. Netw. Exp. Technol. (CoNEXT)*, 2012, pp. 61–72. [Online]. Available: http://doi.acm.org/10.1145/2413176.2413185

[37] L. Rizzo, G. Lettieri, and V. Maffione, "Speeding up packet I/O in virtual machines," in *Proc. Archit. Netw. Commun. Syst.*, San Jose, CA, USA, Oct. 2013, pp. 47–58.

[38] S. Garzarella, G. Lettieri, and L. Rizzo, "Virtual device passthrough for high speed VM networking," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, Oakland, CA, USA, May 2015, pp. 99–110.

[39] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: High performance and flexible networking using virtualization on commodity platforms," *IEEE Trans. Netw. Service Manag.*, vol. 12, no. 1, pp. 34–47, Mar. 2015.

[40] B. Pfaff *et al.*, "Extending networking into the virtualization layer," in *Proc. Workshop Hot Topics Netw. (HotNets-VIII)*, 2009, pp. 1–6.

[41] G. Robin. *Open vSwitch With DPDK Overview. Intel*. Accessed: Apr. 30, 2018. [Online]. Available: https://software.intel.com/en-us/articles/open-vswitch-with-dpdk-overview

[42] *OpenWrt Project*. Accessed: Apr. 30, 2018. [Online]. Available: https://openwrt.org/

[43] *OpenWrt Table of Supported Hardware*. Accessed: Apr. 30, 2018. [Online]. Available: https://openwrt.org/toh/start

[44] D. Medhi and K. Ramasamy, *Network Routing: Algorithms, Protocols, and Architectures*. San Francisco, CA, USA: Morgan Kaufmann, 2007.

[45] C. R. Meiners, A. X. Liu, and E. Torng, *Hardware Based Packet Classification for High Speed Internet Routers*, 1st ed. New York, NY, USA: Springer, 2010. [Online]. Available: https://dl.acm.org/citation.cfm?id=1875239

[46] R. Ramaswamy, N. Weng, and T. Wolf, "Characterizing network processing delay," in *Proc. IEEE Glob. Telecommun. Conf. (GLOBECOM)*, vol. 3. Dallas, TX, USA, Nov. 2004, pp. 1629–1634.

[47] H. Asai, "Where are the bottlenecks in software packet processing and forwarding? Towards high-performance network operating systems," in *Proc. ACM 9th Int. Conf. Future Internet Technol. (CFI)*, 2014, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/2619287.2619300

[48] L. Rizzo, L. Deri, and A. Cardigliano, "10 Gbit/s line rate packet processing using commodity hardware: Survey and new proposals," 2012, pp. 1–8. [Online]. Available: http://luca.ntop.org/10g.pdf

[49] *Scalable Networking Eliminating the Receive Processing Bottleneck—Introducing RSS*, Microsoft WinHEC, Seattle, WA, USA, Apr. 2004.

[50] L. Deri, "nCap: Wire-speed packet capture and transmission," in *Proc. Workshop End End Monitor. Techn. Services*, May 2005, pp. 47–55.

[51] *First the Tick, Now the Tock Next Generation Intel Microarchitecture (Nehalem)*, Intel, Santa Clara, CA, USA, 2009. [Online]. Available: https://goo.gl/KK6DH5

[52] N. Egi *et al.*, "Understanding the packet processing capability of multi-core servers," Intel, Santa Clara, CA, USA, Rep., 2009. [Online]. Available: https://pdfs.semanticscholar.org/2fb0/6b995c83b95ed5d7251ae240e5cf2775111a.pdf

[53] M. Miao, W. Cheng, F. Ren, and J. Xie, "Smart batching: A load-sensitive self-tuning packet I/O using dynamic batch sizing," in *Proc. IEEE 18th Int. Conf. High Perform. Comput. Commun. IEEE 14th Int. Conf. Smart City IEEE 2nd Int. Conf. Data Sci. Syst. (HPCC/SmartCity/DSS)*, Sydney, NSW, Australia, Dec. 2016, pp. 726–733.

[54] L. Rizzo. *OVS: Accelerating the Datapath Through Netmap/VALE*. Accessed: Apr. 30, 2018. [Online]. Available: http://openvswitch.org/support/ovscon2014/18/1630-ovs-rizzo-talk.pdf

[55] T. Marian, K. S. Lee, and H. Weatherspoon, "NetSlices: Scalable multi-core packet processing in user-space," in *Proc. 8th ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, Austin, TX, USA, 2012, pp. 27–38. [Online]. Available: http://doi.acm.org/10.1145/2396556.2396563

[56] L. Deri, "Improving passive packet capture: Beyond device polling," in *Proc. SANE*, 2004, pp. 85–93.

[57] J. C. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," *ACM Trans. Comput. Syst.*, vol. 15, no. 3, pp. 217–252, Aug. 1997. [Online]. Available: http://doi.acm.org/10.1145/263326.263335

[58] J. H. Salim, "When NAPI comes to town," in *Proc. Linux Conf.*, 2005. [Online]. Available: https://archive.org/details/ukuug2005

[59] Intel. *Impressive Packet Processing Performance Enables Greater Workload Consolidation*. Accessed: Apr. 30, 2018. [Online]. Available: https://www.techonline.com/electrical-engineers/education-training/tech-papers/4415080/Impressive-Packet-Processing-Performance-Enables-Greater-Workload-Consolidation

[60] 6WIND. *6WINDGate Packet Processing Software for COTS Servers Scalable Data Plane Performance for Multicore Platforms*. Accessed: Apr. 30, 2018. [Online]. Available: http://www.6wind.com/wp-content/uploads/2016/08/6WINDGate-Data-Sheet.pdf

[61] J. Nickolls, "GPU parallel computing architecture and CUDA programming model," in *Proc. IEEE Hot Chips 19 Symp. (HCS)*, Stanford, CA, USA, Aug. 2007, pp. 1–12.

[62] K. Fatahalian and M. Houston, "A closer look at GPUs," *Commun. ACM*, vol. 51, no. 10, pp. 50–57, Oct. 2008. [Online]. Available: http://doi.acm.org/10.1145/1400181.1400197

[63] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen, "Raising the bar for using GPUs in software packet processing," in *Proc. 12th USENIX Conf. Netw. Syst. Design Implement. (NSDI)*, 2015, pp. 409–423. [Online]. Available: http://dl.acm.org/citation.cfm?id=2789770.2789799

[64] V. Volkov, "Understanding latency hiding on GPUs," Ph.D. dissertation, Dept. EECS, Univ. California, Berkeley, CA, USA, Aug. 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-143.html

[65] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis, "GASPP: A GPU-accelerated stateful packet processing framework," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Philadelphia, PA, USA, 2014, pp. 321–332. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/vasiliadis

[66] Y. Liao, D. Yin, and L. Gao, "PdP: Parallelizing data plane in virtual network substrate," in *Proc. 1st ACM Workshop Virtual. Infrastruct. Syst. Archit. (VISA)*, Barcelona, Spain, 2009, pp. 9–18. [Online]. Available: http://doi.acm.org/10.1145/1592648.1592651

[67] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, "Comparison of frameworks for high-performance packet IO," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, Oakland, CA, USA, May 2015, pp. 29–38.

[68] T. Hudek and D. MacMichael. *Introduction to the NDIS PacketDirect Provider Interface*. Accessed: Apr. 30, 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-ndis-pdpi

[69] S. Gueron, "Speeding up CRC32C computations with Intel CRC32 instruction," *Inf. Process. Lett.*, vol. 112, no. 5, pp. 179–185, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020019011003l9X

[70] Intel. *Intel® SSE4 Programming Reference, p.61*. Accessed: Apr. 30, 2018. [Online]. Available: https://software.intel.com/sites/default/files/m/8/b/8/D9156103.pdf

[71] L. Rizzo, M. Carbone, and G. Catalli, "Transparent acceleration of software packet forwarding using netmap," in *Proc. IEEE INFOCOM*, Orlando, FL, USA, Mar. 2012, pp. 2471–2479.

[72] D. Blythe, "Rise of the graphics processor," *Proc. IEEE*, vol. 96, no. 5, pp. 761–778, May 2008.

[73] J. D. Owens *et al.*, "A survey of general-purpose computation on graphics hardware," *Comput. Graph. Forum*, vol. 26, no. 1, pp. 80–113, 2007. [Online]. Available: http://dx.doi.org/10.1111/j.1467-8659.2007.01012.x

[74] J. Tseng *et al.*, "Exploiting integrated GPUs for network packet processing workloads," in *Proc. IEEE NetSoft Conf. Workshops (NetSoft)*, Seoul, South Korea, Jun. 2016, pp. 161–165.

[75] E. Papadogiannaki, L. Koromilas, G. Vasiliadis, and S. Ioannidis, "Efficient software packet processing on heterogeneous and asymmetric hardware architectures," *IEEE/ACM Trans. Netw.*, vol. 25, no. 3, pp. 1593–1606, Jun. 2017.

[76] Z. Zheng, J. Bi, H. Yu, C. Sun, and J. Wu, "BLOP: Batch-level order preserving for GPU-accelerated packet processing," in *Proc. ACM SIGCOMM Posters Demos (SIGCOMM)*, Los Angeles, CA, USA, 2017, pp. 136–137. [Online]. Available: http://doi.acm.org/10.1145/3123878.3132013

[77] F. Fusco, M. Vlachos, X. Dimitropoulos, and L. Deri, "Indexing million of packets per second using GPUs," in *Proc. ACM Conf. Internet Meas. Conf. (IMC)*, Barcelona, Spain, 2013, pp. 327–332. [Online]. Available: http://doi.acm.org/10.1145/2504730.2504756

[78] J. D. Owens *et al.*, "GPU computing," *Proc. IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[79] M. Garland, "Parallel computing with CUDA," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. (IPDPS)*, Apr. 2010, p. 1.

[80] Y. Zhu, Y. Deng, and Y. Chen, "Hermes: An integrated CPU/GPU microarchitecture for IP routing," in *Proc. 48th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, New York, NY, USA, Jun. 2011, pp. 1044–1049.

[81] S. Mu *et al.*, "IP routing processing with graphic processors," in *Proc. Design Autom. Test Europe Conf. Exhib. (DATE)*, Dresden, Germany, Mar. 2010, pp. 93–98.

[82] B. Betkaoui, D. B. Thomas, and W. Luk, "Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing," in *Proc. Int. Conf. Field Program. Technol. (FPT)*, Beijing, China, Dec. 2010, pp. 94–101.

[83] S. Mittal and J. S. Vetter, "A survey of methods for analyzing and improving GPU energy efficiency," *ACM Comput. Surveys*, vol. 47, no. 2, pp. 1–23, Aug. 2014. [Online]. Available: http://doi.acm.org/10.1145/2636342

[84] D. Bacon, R. Rabbah, and S. Shukla, "FPGA programming for the masses," *Queue*, vol. 11, no. 2, pp. 40–52, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2436696.2443836

[85] T. Rinta-Aho, M. Karlstedt, and M. P. Desai, "The Click2NetFPGA toolchain," in *Proc. USENIX Conf. Annu. Tech. Conf. (USENIX ATC)*, 2012, p. 7. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342821.2342828

[86] G. Gibb, J. W. Lockwood, J. Naous, P. Hartke, and N. McKeown, "NetFPGA—An open platform for teaching how to build gigabit-rate network switches and routers," *IEEE Trans. Edu.*, vol. 51, no. 3, pp. 364–369, Aug. 2008.

[87] J. W. Lockwood *et al.*, "NetFPGA–An open platform for gigabit-rate network switching and routing," in *Proc. IEEE Int. Conf. Microelectron. Syst. Educ. (MSE)*, San Diego, CA, USA, 2007, pp. 160–161, doi: 10.1109/MSE.2007.69.

[88] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "NetFPGA SUME: Toward 100 Gbps as research commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, Sep./Oct. 2014, doi: 10.1109/MM.2014.61.

[89] N. Zilberman *et al.*, "NetFPGA—Rapid prototyping of high bandwidth devices in open source," in *Proc. 25th Int. Conf. Field Program. Logic Appl. (FPL)*, Sep. 2015, p. 1.

[90] M. Labrecque, J. G. Steffan, G. Salmon, M. Ghobadi, and Y. Ganjali, "NetThreads: Programming NetFPGA with threaded software," in *Proc. NetFPGA Developers Workshop*, Aug. 2009. [Online]. Available: https://www.microsoft.com/en-us/research/publication/netthreads-programming-netfpga-threaded-software/

[91] P. Varga and L. Kovács and T. Tóthfalusi and P. Orosz, "C-GEP: 100 Gbit/s capable, FPGA-based, reconfigurable networking equipment," in *Proc. IEEE 16th Int. Conf. High Perform. Switching Routing (HPSR)*, Jul. 2015, pp. 1–6.

[92] Y. O. M. Moctar, G. G. F. Lemieux, and P. Brisk, "Fast and memory-efficient routing algorithms for field programmable gate arrays with sparse intracluster routing crossbars," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 34, no. 12, pp. 1928–1941, Dec. 2015.

[93] D. F. G. Prado, *Tutorial on FPGA Routing*, Electrónica-UNMSM, Lima, Peru, no. 17, pp. 23–33, 2006.

[94] V. G. Gudise and G. K. Venayagamoorthy, "FPGA placement and routing using particle swarm optimization," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI*, Feb. 2004, pp. 307–308.

[95] I. Kuon, R. Tessier, and J. Rose, "FPGA architecture: Survey and challenges," *Found. Trends® Electron. Design Autom.*, vol. 2, no. 2, pp. 135–253, 2008. [Online]. Available: http://dx.doi.org/10.1561/1000000005

[96] L. McMurchie and C. Ebeling, "PathFinder: A negotiation-based performance-driven router for FPGAs," in *Proc. 3rd Int. ACM Symp. Field Program. Gate Arrays*, 1995, pp. 111–117.

[97] M. Stojilović, "Parallel FPGA routing: Survey and challenges," in *Proc. 27th Int. Conf. Field Program. Logic Appl. (FPL)*, Sep. 2017, pp. 1–8.

[98] P. K. Chan and M. D. F. Schlag, "Acceleration of an FPGA router," in *Proc. 5th Annu. IEEE Symp. Field Program. Custom Comput. Mach.*, Apr. 1997, pp. 175–181.

[99] Y.-W. Chang, S. Thakur, K. Zhu, and D. F. Wong, "A new global routing algorithm for FPGAs," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, San Jose, CA, USA, Nov. 1994, pp. 356–361.

[100] G. R. M. J. Alexander, "New performance-driven FPGA routing algorithms," in *Proc. 32nd Design Autom. Conf.*, San Francisco, CA, USA, 1995, pp. 562–567.

[101] B. Wheeler, "A new era of network processing," Mountain View, CA, USA, Linley Group, White Paper, 2013.

[102] J. Shim, J. Kim, K. Lee, and S. Moon, "Knapp: A packet processing framework for manycore accelerators," in *Proc. IEEE 3rd Int. Workshop High Perform. Interconnection Netw. Exascale Big Data Era (HiPINEB)*, Austin, TX, USA, Feb. 2017, pp. 57–64.

[103] G. Chrysos, "Intel® Xeon Phi coprocessor (codename Knights Corner)," in *Proc. IEEE Hot Chips 24 Symp. (HCS)*, Aug. 2012, pp. 1–31.

[104] D. Cerovic, V. D. Piccolo, A. Amamou, and K. Haddadou, "Offloading TRILL on a programmable card," in *Proc. 3rd Smart Cloud Netw. Syst. (SCNS)*, Dec. 2016, pp. 1–5.

[105] *Click Modular Router*. Accessed: Apr. 30, 2018. [Online]. Available: https://github.com/kohler/click/

[106] *RouteBricks*. Accessed: Apr. 30, 2018. [Online]. Available: http://routebricks.org/code.html

[107] *FastClick*. Accessed: Apr. 30, 2018. [Online]. Available: https://gitlab.run.montefiore.ulg.ac.be/sdn-pp/fastclick/

[108] *Snap*. Accessed: Apr. 30, 2018. [Online]. Available: https://github.com/wbsun/snap

[109] *Netmap*. Accessed: Apr. 30, 2018. [Online]. Available: https://github.com/luigirizzo/netmap

[110] *NetSlice*. Accessed: Apr. 30, 2018. [Online]. Available: http://fireless.cs.cornell.edu/netslice/

[111] *PF_RING*. Accessed: Apr. 30, 2018. [Online]. Available: https://github.com/ntop/PF_RING

[112] *DPDK*. Accessed: Apr. 30, 2018. [Online]. Available: http://dpdk.org/download
[113] *PacketShader*. Accessed: Apr. 30, 2018. [Online]. Available: https://github.com/ANLAB-KAIST/Packet-IO-Engine
[114] *PacketShader Website*. Accessed: Apr. 30, 2018. [Online]. Available: https://shader.kaist.edu/packetshader/
[115] M. Bourguiba, K. Haddadou, and G. Pujolle, "A container-based fast bridge for virtual routers on commodity hardware," in *Proc. IEEE Glob. Telecommun. Conf. (GLOBECOM)*, Miami, FL, USA, Dec. 2010, pp. 1–6.
[116] M. Bourguiba, K. Haddadou, I. E. Korbi, and G. Pujolle, "A container-based I/O for virtual routers: Experimental and analytical evaluations," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Jun. 2011, pp. 1–6.
[117] P. Barham *et al.*, "Xen and the art of virtualization," in *Proc. 19th ACM Symp. Oper. Syst. Principles (SOSP)*, 2003, pp. 164–177. [Online]. Available: http://doi.acm.org/10.1145/945445.945462
[118] M. Bourguiba, K. Haddadou, I. E. Korbi, and G. Pujolle, "Improving network I/O virtualization for cloud computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 3, pp. 673–681, Mar. 2014.
[119] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. USENIX Annu. Tech. Conf. FREENIX Track*, 2005, pp. 41–46.
[120] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: The Linux virtual machine monitor," in *Proc. Ottawa Linux Symp. (OLS)*, 2007, pp. 225–230.
[121] V. Maffione, L. Rizzo, and G. Lettieri, "Flexible virtual machine networking using netmap passthrough," in *Proc. IEEE Int. Symp. Local Metropolitan Area Netw. (LANMAN)*, Jun. 2016, pp. 1–6.
[122] R. Russell, "Virtio: Towards a de-facto standard for virtual I/O devices," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 95–103, Jul. 2008. [Online]. Available: http://doi.acm.org/10.1145/1400097.1400108
[123] *Open vSwitch* Enables SDN and NFV Transformation*, Intel, Santa Clara, CA, USA, 2015. [Online]. Available: https://networkbuilders.intel.com/docs/open-vswitch-enables-sdn-and-nfv-transformation-paper.pdf
[124] J. Tseng, R. Wang, J. Tsai, Y. Wang, and T.-Y. C. Tai, "Accelerating open vSwitch with integrated GPU," in *Proc. Workshop Kernel Bypass Netw. (KBNets)*, Los Angeles, CA, USA, 2017, pp. 7–12. [Online]. Available: http://doi.acm.org/10.1145/3098583.3098585
[125] R. Kawashima, S. Muramatsu, H. Nakayama, T. Hayashi, and H. Matsuo, "A host-based performance comparison of 40G NFV environments focusing on packet processing architectures and virtual switches," in *Proc. 5th Eur. Workshop Softw. Defined Netw. (EWSDN)*, Oct. 2016, pp. 19–24.
[126] E. Kohler, "The click modular router," Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., Massachusetts Inst. Technol., Cambridge, MA, USA, 2001.
[127] *Altera SDK for OpenCL*. Accessed: Apr. 30, 2018. [Online]. Available: http://www.altera.com/
[128] *SDAccel Development Environment*. Accessed: Apr. 30, 2018. [Online]. Available: http://www.xilinx.com/
[129] *Vivado Design Suite*. Accessed: Apr. 30, 2018. [Online]. Available: http://www.xilinx.com/
[130] *NVIDIA CUDA SDK 3.0*. Accessed: Apr. 30, 2018. [Online]. Available: https://developer.nvidia.com/cuda-toolkit-30-downloads
[131] *Intel—Linux Ixgbe* Base Driver Overview and Installation*. Accessed: Apr. 30, 2018. [Online]. Available: https://www.intel.com/content/www/us/en/support/network-and-i-o/ethernet-products/000005688.html
[132] A. Putnam *et al.*, "A Reconfigurable fabric for accelerating large-scale datacenter services," in *Proc. 41st Annu. Int. Symp. Comput. Archit. (ISCA)*, Minneapolis, MN, USA, 2014, pp. 13–24. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665678
[133] *Syskonnect Gigabit Ethernet Card*. Accessed: Apr. 30, 2018. [Online]. Available: http://www.syskonnect.com/
[134] *XMACII Chipset*. Accessed: Apr. 30, 2018. [Online]. Available: https://goo.gl/zUrcnr
[135] *OpenFastPath–Technical Overview*. Accessed: Apr. 30, 2018. [Online]. Available: http://www.openfastpath.org/index.php/service/technicaloverview/
[136] J. Corbet and G. Kroah-Hartman, *Linux Kernel Development: How Fast It Is Going, Who Is Doing It, What They Are Doing, and Who Is Sponsoring the Work*, Linux Found., San Francisco, CA, USA, pp. 1–19, 2016.
[137] *FD.io*. Accessed: Apr. 30, 2018. [Online]. Available: https://fd.io/
[138] E. Warnicke *FD.io Intro*. Accessed: Apr. 30, 2018. [Online]. Available: https://wiki.fd.io/view/File:Fdio_intro_2016-03-10.pptx
[139] *Vector Packet Processing (VPP)*. Accessed: Apr. 30, 2018. [Online]. Available: https://wiki.fd.io/view/VPP
[140] *VPP/What Is VPP?* Accessed: Apr. 30, 2018. [Online]. Available: https://goo.gl/q1Jqhw
[141] *VPP on OpenDataPlane Enabled SmartNICs*. Accessed: Apr. 30, 2018. [Online]. Available: goo.gl/EV1Qep
[142] *OpenDataPlane (ODP) Project*. Accessed: Apr. 30, 2018. [Online]. Available: https://www.opendataplane.org/
[143] E. Warnicke. *OpenDataPlane (ODP) Users-Guide*. Accessed: Apr. 30, 2018. [Online]. Available: https://www.opendataplane.org/application-writers/users-guide/
[144] *PCIe NIC Optimised Implementation (ODP-DPDK)*. Accessed: Apr. 30, 2018. [Online]. Available: https://git.linaro.org/lng/odp-dpdk.git
[145] *OpenFastPath*. Accessed: Apr. 30, 2018. [Online]. Available: http://www.openfastpath.org/
[146] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2656877.2656890
[147] H. Wang *et al.*, "P4FPGA: A rapid prototyping framework for P4," in *Proc. Symp. SDN Res. (SOSR)*, Santa Clara, CA, USA, 2017, pp. 122–135. [Online]. Available: http://doi.acm.org/10.1145/3050220.3050234
[148] P. Li and Y. Luo, "P4GPU: Accelerate packet processing of a P4 program with a CPU-GPU heterogeneous architecture," in *Proc. Symp. Archit. Netw. Commun. Syst. (ANCS)*, Santa Clara, CA, USA, 2016, pp. 125–126. [Online]. Available: http://doi.acm.org/10.1145/2881025.2889480
[149] P. Benácek, V. Pu, and H. Kubátová, "P4-to-VHDL: Automatic generation of 100 Gbps packet parsers," in *Proc. IEEE 24th Annu. Int. Symp. Field Program. Custom Comput. Mach. (FCCM)*, Washington, DC, USA, May 2016, pp. 148–155.
[150] A. Sivaraman, C. Kim, R. Krishnamoorthy, A. Dixit, and M. Budiu, "DC.P4: Programming the forwarding plane of a data-center switch," in *Proc. 1st ACM SIGCOMM Symp. Softw. Defined Netw. Res. (SOSR)*, Santa Clara, CA, USA, 2015, pp. 1–8. [Online]. Available: [Online]. Available: http://doi.acm.org/10.1145/2774993.2775007
[151] S. Laki *et al.*, "High speed packet forwarding compiled from protocol independent data plane specifications," in *Proc. ACM SIGCOMM Conf. (SIGCOMM)*, Florianópolis, Brazil, 2016, pp. 629–630. [Online]. Available: http://doi.acm.org/10.1145/2934872.2959080
[152] A. Bhardwaj, A. Shree, V. B. Reddy, and S. Bansal, "A preliminary performance model for optimizing software packet processing pipelines," in *Proc. 8th Asia–Pac. Workshop Syst. (APSys)*, Mumbai, India, 2017, pp. 1–7. [Online]. Available: http://doi.acm.org/10.1145/3124680.3124747
[153] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "OpenState: Programming platform-independent stateful OpenFlow applications inside the switch," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 44–51, Apr. 2014. [Online]. Available: http://doi.acm.org/10.1145/2602204.2602211
[154] S. Pontarelli, M. Bonola, G. Bianchi, A. Capone, and C. Cascone, "Stateful OpenFlow: Hardware proof of concept," in *Proc. IEEE 16th Int. Conf. High Perform. Switching Routing (HPSR)*, Jul. 2015, pp. 1–8.
[155] *BESS (Berkeley Extensible Software Switch)*. Accessed: Apr. 30, 2018. [Online]. Available: http://span.cs.berkeley.edu/bess.html
[156] S. Han *et al.*, "SoftNIC: A software NIC to augment hardware," EECS Dept., Univ. California, Berkeley, CA, USA, Rep. UCB/EECS-2015-155, May 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html

Danilo Cerović received the M.S. degree in advanced wireless communications from CentraleSupélec, Gif-sur-Yvette, France, in 2015, and the M.S. degree in systems engineering and radiocommunications from the School of Electrical Engineering, University of Belgrade, Belgrade, Serbia, in 2015. He is currently pursuing the Ph.D. degree with Sorbonne Université (UPMC—LIP6) and GANDI SAS.

Valentin Del Piccolo received the M.S. and Ph.D. degrees in networking and computer science from University Pierre and Marie Curie, Paris, France, in 2013 and 2017, respectively. He is a Research Engineer with GANDI SAS.

Ahmed Amamou (M'13) received the Engineering degree in computer science and the M.S. degree in network and computer science from the National School of Computer Science, Tunisia, in 2009 and 2011, and the Ph.D. degree in network and computer science from the University Pierre and Marie Curie, Paris, France, in 2013. He is a Research Engineer with GANDI SAS. His research interests are cloud computing and virtualization technologies.

Kamel Haddadou (M'18) received the Engineering degree in computer science from INI in 2000, the M.S. degree in data processing methods for industrial systems from the University of Versailles, and the Ph.D. degree in computer networks from University Pierre and Marie Curie, in 2002 and 2007, respectively. In 2001, he was a Research Assistant with the Advanced Technology Development Centre, Algiers, Algeria. He is currently a Research Fellow with Gandi SAS, France. Since 2003, he has been involved in several projects funded by the European Commission and the French Government (RAVIR, ADANETS, Adminroxy, GITAN, OGRE, ADANETS, MMQoS, SAFARI, and ARCADE). His research interests are focused primarily on cloud computing and on resource management in wired and wireless networks. He is equally interested in designing new protocols and systems with theoretical concepts, and in providing practical implementations that are deployable in real environments. He has served as the TPC member for many international conferences, including IEEE ICC, GLOBECOM, and a reviewer on a regular basis for major international journals and conferences in networking.

Guy Pujolle received the Ph.D. and These d'Etat degrees in computer science from the University of Paris IX and Paris XI in 1975 and 1978, respectively. He is currently a Professor with Sorbonne Université (UPMC—LIP6), Paris, France. He is the French Representative with Technical Committee on Networking, IFIP. He is an Editor for the *ACM International Journal of Network Management*, *Telecommunication Systems*, and the Editor-in-Chief of the *Annals of Telecommunications*. He is a pioneer in high-speed networking having led the development of the first Gbit/s network to be tested in 1980. He has participated in several important patents like DPI or virtual networks. He is the Co-Founder of QoSMOS, Ucopia Communications, EtherTrust, and Green Communications.