Resilient and highly performant network architecture for virtualized data centers

Danilo Cerović

THÈSE DE DOCTORAT DE
LA SORBONNE UNIVERSITÉ

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique de Paris

Présentée par

Danilo CEROVIĆ

Pour obtenir le grade de

DOCTEUR de la SORBONNE UNIVERSITE

Sujet de la thèse :

Architecture réseau résiliente et hautement performante pour les datacenters virtualisés

soutenue le 7 février 2019
devant le jury composé de :

M. Raouf BOUTABA	Rapporteur	Professeur - Université de Waterloo
M. Olivier FESTOR	Rapporteur	Directeur - Télécom Nancy
M. Djamal ZEGHLACHE	Examinateur	Professeur - Télécom SudParis
M. Christian JACQUENET	Examinateur	Referent Expert, Networks of the Future - Orange Labs
Mme. Michelle SIBILLA	Examinateur	Professeur - IRIT
M. Marcelo DIAS DE AMORIM	Examinateur	Directeur de Recherche - CNRS
M. Guy PUJOLLE	Directeur de thèse	Professeur - Sorbonne Université
M. Kamel HADDADOU	Encadrant	Directeur de Recherche et Développement - Gandi

À la mémoire de mon père

Abstract

The amount of traffic in data centers is growing exponentially and it is not expected to stop growing any time soon. This brings about a vast amount of advancements in the networking field. Network interface throughputs supported today are in the range of 40Gbps and higher. On the other hand, such high interface throughputs do 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. The networking stack of an operating system is not conceived for high-speed networking applications but rather for general purpose communications.

In this thesis, we investigate various approaches that strive to improve packet processing performance on server-class network hosts, either by using software, hardware, or the combination of the two. Some of the solutions are based on the Click modular router which offloads its functions on different types of hardware like GPUs, FPGAs 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 software and hardware packet processing solutions based on different criteria and we discuss their integration possibilities in virtualized environments, their constraints and their requirements.

As our first contribution, we propose a resilient and highly performant fabric network architecture. Our goal is to build a layer 2 mesh network that only uses directly connected hardware acceleration cards that perform packet processing instead of routers and switches. We have decided to use the TRILL protocol for the communication between these smart NICs as it provides a better utilization of network links while also providing least-cost pair-wise data forwarding. The data plane packet processing is offloaded on a programmable hardware with parallel processing capability. Additionally, we propose to use the ODP API so that packet processing application code can be reused by any other packet processing solution that supports the ODP API.

As our second contribution, we designed a data plane of the TRILL protocol on the MPPA (Massively Parallel Processor Array) smart NIC which supports the ODP API. Our experimental results show that we can process TRILL frames at full-duplex line-rate (up to 40Gbps) for different packet sizes while reducing latency.

As our third contribution, we provide a mathematical analysis of the impact of different network topologies on the control plane's load. The data plane packet processing is performed on the MPPA smart NICs. Our goal is to build a layer 2 mesh network that only uses directly connected smart NIC cards instead of routers and switches. We have considered various network topologies and we compared their loads induced by the control plane traffic. We have also shown that hypercube topology is the most suitable for our PoP data center use case because it does not have a high control plane load and it has a better resilience than fat-tree while having a shorter average distance between the nodes.

Keywords

Networking, Packet processing, Cloud computing, ODP, MPPA, TRILL, Virtualization, DPDK.

Table of contents

List of Figures

List of Tables

Chapter 1

Introduction

Summary

1.1 Motivation

Packet processing represents a large set of algorithms which are applied to a packet of information or data which is transferred between different network elements. As the throughputs supported by network interfaces are getting higher, there is a corresponding need for faster packet processing. Forwarding engine is the most important part of a high-speed router and with the exponential growth of Internet traffic, which is not expected to slow down [20], there is a high demand for the introduction of multi-terabit IP routers [21]. Similarly, inside the data centers 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 [22] and Broadcom Tomahawk 3 [23] and they are used to aggregate a large number of high-speed links. Nevertheless, packet processing speeds in software can hardly follow the increased throughput supported by today's network interfaces. The reason behind this is the generality of the network stack's architecture and the device driver design that bound the traffic processing rate despite the advances in NICs (Network Interface Cards) development. In section 2 we investigate fast packet processing solutions on server-class network hosts that try to overcome this limitation 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.

Servers typically have network interfaces with speeds of 40Gbps [24] in today's data centers. However, the network stacks' architectural design comes with a high-performance cost which immensely limits the effectiveness of increasingly powerful hardware [25]. The overheads that it implies represent a key obstacle to effective scaling. In fact, in a modern OS (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 [26].

Packet processing over multiple 10Gbps interfaces at line-rate can hardly be supported by a standard general-purpose kernel stack. For this reason, many packet processing solutions that significantly improve network performance have been proposed and they are based either on hardware, software or their hybrid combination. Different user-space I/O frameworks allow bypassing the kernel and obtaining a batch of raw packets through a single syscall. Additionally, they support modern Network Interface Card (NIC) capabilities such as multi-queueing [27]. Namely, Intel's DPDK [28], Netmap [4], PacketShader [8], PF_RING [6], OpenOnload [29], PACKET_MMAP [30] are examples of such frameworks out of which most relevant ones, in terms of their widespread usage, are reviewed in this thesis among other solutions. There are many solutions that are based on the Click Modular Router [3] which is a software architecture that is used for building flexible and configurable routers. Some of the most notable ones are: RouteBricks [31], FastClick [27], ClickNP [13], Snap [32], DoubleClick [33], SMP Click [34], NP-Click [35]. Majority of these solutions aim to improve Click's performance by allowing to parallelize the packet processing workload on multicore systems.

There are two possible approaches to build programmable network infrastructures with high performance, as has been discussed in [31]. One is to start from high-end specialized equipment and make it programmable such as the idea of OpenFlow [36, 37]. The other begins with software routers and optimizes their packet processing performance. This second approach allows programmers to work in the well-known environments of general-purpose computers. Nonetheless, there are "semi-specialized" solutions which use network processors to offload a part of the packet processing that represents a performance bottleneck. The rest of the packet processing is done on conventional processors. Examples of solutions which use NPs (Network Processors) are NP-Click [35] along with works such as [38] in which the authors implement OpenFlow switch over the EZchip NP4 Network Processor. In [39] various programming techniques and optimization issues that can significantly improve the performance of Network Processors are investigated. Furthermore, a profound understanding of NP design trade-offs by using an analytic performance model configured by realistic workload and technology parameters is given in [40]. At the same time, certain solutions use reconfigurable hardware (Field Programmable Gate Arrays - FPGAs) as they are more versatile compared to NPs. Namely, Chimpp [41], ClickNP [13], SwitchBlade [11], GRIP [10], Netfpga [42] along with other works in this area [43, 44]. The most well-known solutions which use Graphics Processing Units (GPUs) for packet processing are certainly PacketShader [8], Snap [32], APUNet [9] and GASPP [45].

The underlying hardware such as a CPU, GPU or an FPGA has a major impact on the achievable packet processing performance of each solution. Each hardware type has its own advantages and dis-

advantages. 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 not easily programmable as they use HDL (Hardware Description Language) but they are more energy efficient than GPUs and CPUs. Moreover, FPGAs provide deterministic timing where latencies are one order of magnitude lower when compared with GPUs [46]. This feature is crucial for hard real-time tasks and applications like network synchronization, encryption, etc. More details on this topic are given in section 3.5.

The majority 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 [47] solution. Furthermore, in [48] the packet I/O speed in the VMs is further improved, while ptnetmap [49] is the Virtual Passthrough solution based on the netmap framework. Similarly, there are DPDK-based solutions that also aim to accelerate the packet processing speed for VMs. Namely, NetVM [50] 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 [51]) as in OvS-DPDK [52] 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, while OvS-DPDK improves its performance.

The data center network infrastructure interconnects end devices (i.e. servers) inside a data center or across data centers [53]. Nowadays, VDC's (Virtualized Data Center) physical servers are typically divided into multiple mutually isolated virtual instances. Each instance can be running a different OS and can use its own configuration set (CPU, disk space, memory size). An increasing number of such virtual instances needs to be hosted on a VDC [1] which brings about new challenging requirements such as: any-to-any connectivity with full non-blocking fabric, dynamic scalability, fault tolerance, seamless VM (Virtual Machine) mobility, simplified management and private network support.

A protocol that is able to fulfill such requirements is needed. One such protocol is TRILL (TRansparent Interconnection of Lots of Links) [54] and it has been conceived by IETF (Internet Engineering Task Force). A more detailed introduction to TRILL protocol can be found in Appendix A. TRILL has a simple configuration and deployment like Ethernet and it brings certain advantages of layer 3 routing techniques to layer 2. It is also considered as a layer 2 and 1/2 protocol since it terminates traditional Ethernet clouds, similar to what IP routers do. Furthermore, a conversion from existing Ethernet deployments into a TRILL network can be done by replacing classical bridges with RBridges (Routing Bridges) which implement TRILL protocol. TRILL is transparent to layer 3 protocols.

RBridges perform the SPF (Shortest Path First) algorithm, thus providing optimal pair-wise forwarding. TRILL provides multipathing for both unicast and multicast traffic [54] which enables better link utilization in the network by using multiple paths to transfer data between the two end hosts. This

is enabled by the encapsulation of traffic with an additional TRILL header that includes the hop count field. This field is characteristic of layer 3 protocols and allows for safe forwarding during periods of temporary loops. RBridges are invisible to IP routers in the same way that IEEE 802.1 customer bridges are. Additionally, RBridges are backward-compatible with IEEE 802.1 customer bridges [54]. Whenever an Ethernet frame encounters the first RBridge (called the ingress RBridge) on its way to a destination, it gets encapsulated. The encapsulation process consists of adding a TRILL header and an outer Ethernet header around the original frame with its (inner) Ethernet header. The ingress RBridge specifies in the TRILL header the RBridge closest to the final destination (called the egress RBridge) to which the frame should be forwarded. Afterward, the frame gets forwarded towards the egress RBridge while on each RBridge on the path the received frame needs to be analyzed and processed. On each hop, the outer Ethernet header is set according to the next hop in direction of the egress RBridge. Additionally, the hop count field in the TRILL header is decremented. This frame processing represents the communication bottleneck between servers inside a data center as it can impact significantly the frame forwarding rate. The problem has been identified in the existing data plane implementation of TRILL protocol in a modified Linux Bridge Module [1]. This network performance degradation also represents one of the main problems to be solved in this thesis.

1.2 Problematics

Cloud service providers still widely use Ethernet technology in order to interconnect the servers inside a data center because of Ethernet's simple operation and configuration. Ethernet bridges automatically learn the MAC addresses of the hosts that are connected to it by associating, in the forwarding table, the receiving port of the bridge with the source MAC address of the frame that arrived on that port. However, if the destination MAC address is not known in the forwarding table, the frame gets broadcast over all the ports of the bridge except the one that received a frame. Additionally, Ethernet frames do not have a hop count field that could limit the number of hops of the frame inside a data center. Consequently, any loop in the network could cause a serious failure in a data center's functionality because of the replication of frames that are directed to an unknown destination and another replication in the case of the loop. Propagation of this problem continues until the frame forwarding capacity of the bridge is reached and link failures start to occur. For this reason, STP (Spanning Tree Protocol) has been introduced and it allows to build a loop-free logical topology in the form of a tree which forms a single communication path for any pair of nodes in the network. Inside such a network, any VM migration is possible because of the unique MAC address that each node has.

Nevertheless, in cloud environments there is a scaling problem when tens of thousands of VMs are present in a single data center. In order to create forwarding tables, network bridges broadcast frames across the network and this flooding causes the overload of the bridges' forwarding tables. In fact, the CAM (Content Addressable Memory) of the layer 2 bridge is very responsive but at the same very expensive which makes it have a limited amount of memory. Consequently, it would be too expensive to replace all the bridges inside a data center with the ones that have CAM memories large enough to hold tens of thousands of VM's MAC addresses or even more. Additionally, STP cuts off a significant number of links in the network and makes them unusable. This makes a majority of communication

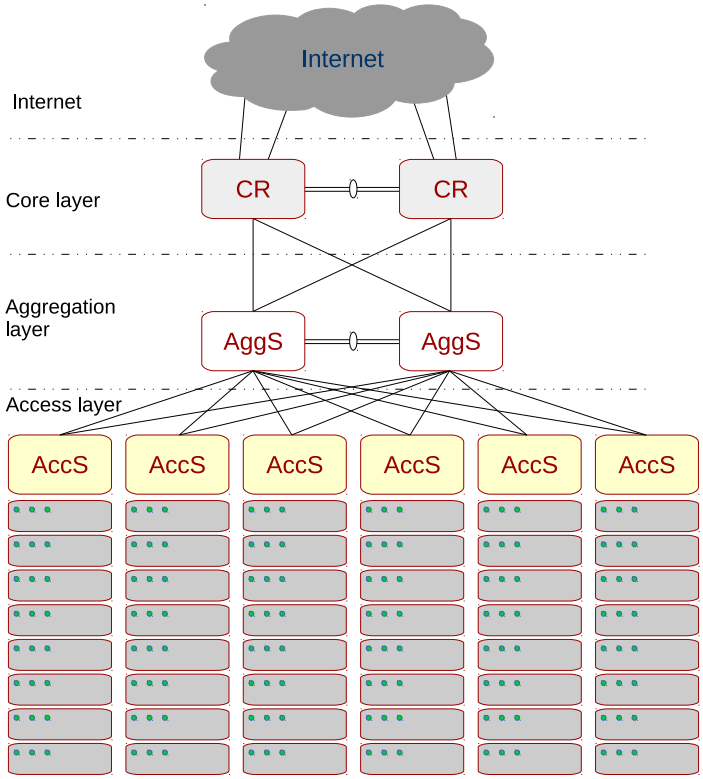


Figure 1.1: Traditional data center network architecture. [1]

between the VMs be done over the same links that belong to the spanning tree. However, there is a rising need for higher throughputs inside a data center which often requires the existence of multiple paths between the bridges. For all these reasons, it is hardly possible to create a topology that has only one layer 2 segment for the whole data center because a high number of nodes may create a congestion in the network.

In traditional data center networks, the adopted standard architecture consists of connecting a set of physical servers that belong to the same rack on a ToR (Top of the Rack) switch. Also, all the ToR switches are connected to the AggSs (Aggregation Switches) that belong to the aggregation layer. This layer aggregates all the traffic from the ToR switches and it also brings redundant paths between ToR switches and the core layer. This architecture is illustrated in Figure 1.1 in which AccSs (Access Switches) correspond to ToR switches. The highest layer in the architecture has CRs (Core Routers) that connect the data center with the outside world and they operate at layer 3 of the OSI reference model. In the virtualized data center, the VMs are hosted on the servers that are connected to ToR switches (AccS switches in Figure 1.1). In this type of architecture, whenever a VM is migrated from one physical server to another one that belongs to a different layer 2 segment, its IP address needs to be changed. Moreover, the STP eliminates a possibility to use multipath communication between the servers. For this reason, we adopt a full-mesh non-blocking architecture that is shown in Figure 1.2. In this case, all the RBridges are directly connected and through the use of TRILL protocol they are

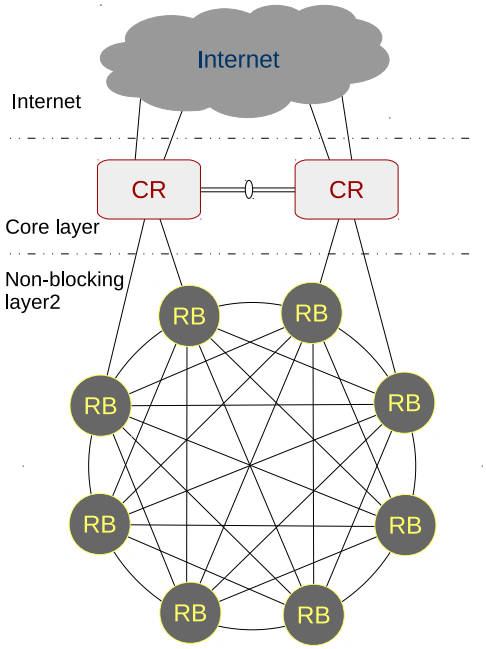


Figure 1.2: TRILL-based non-blocking layer-2 network. [1]

able to use multipathing. On each server that hosts VMs, there will be an RBridge implementation that will provide more efficient communications between VMs that are hosted on different servers. The number of hops needed for communication is significantly reduced and the resilience is improved when compared with a traditional data center architecture.

However, in order to overcome the frame processing bottleneck, presented in the previous section, we have chosen to offload frame processing on the Kalray MPPA (Massively Parallel Processor Array) smart NIC which is power efficient. It has 256 user-programmable cores and allows developing in C programming language. Additionally, it provides the ODP (OpenDataPlane) API which allows developers to create portable networking data plane applications. Despite the fact that the raw computational power of the Kalray MPPA smart NIC may theoretically speed up the processing of TRILL frames, the problem is how to organize the processing among clusters that contain many cores. Additionally, since this smart NIC does not have any networking drivers or networking stack, everything needs to be developed from scratch. The remaining question is to verify whether the full-mesh topology is the best-suited for all data center sizes and to compare the load induced by the control plane traffic of different topologies. More specifically, we were interested in the use case of PoP (Point of Presence) data centers with a limited number of servers when MPPA smart NICs are used.

1.3 Contributions

The goal of this thesis is to propose a resilient and highly performant network architecture for

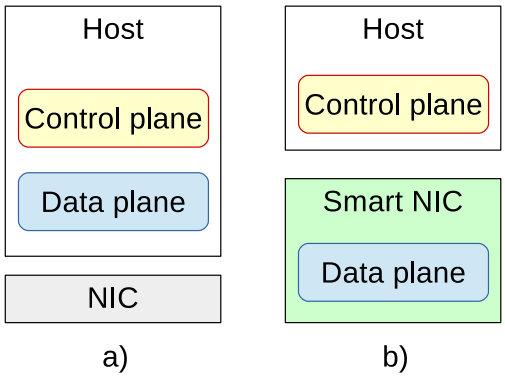


Figure 1.3: Implementation that performs data plane packet processing inside a) a Linux Bridge Module and b) a smart NIC.

virtualized data centers that uses Kalray MPPA smart NIC acceleration cards instead of routers and switches in order to overcome the performance bottleneck of the OS network stack.

As our first contribution, we have proposed an architecture that aims to overcome the limitations of the existing solution [1] that is provisioned in the public platform of Gandi SAS. The data plane is implemented in a modified Linux Bridge Module (open-source code available in [55]) as shown in Figure 1.3.a). This allowed the creation of a full mesh non-blocking layer 2 network based on TRILL protocol which does not need to use switches, as shown in Figure 1.2. RBs represent the RBridges, while CRs represent core layer routers. Nevertheless, packet processing has limited performance in Linux because of the generality of the network stack architecture and generality of the device driver design that affect the network throughput. Our idea is to build a fabric network that is highly performant and that offloads the data plane packet processing tasks on hardware acceleration cards instead of performing them in the Linux kernel on servers' CPUs (as shown in Figure 1.3.b).). The plan is to replace the classical three-tier network topology with the full-mesh network of links which allows to have predictable network performance and direct connectivity between the nodes. Furthermore, in our third contribution, we consider other network topologies in order to determine which one is best-suited for our PoP data center use case. A flat architecture that we propose is scalable, resilient and generally decreases the number of hops needed for communication between the servers. We have decided to use the TRILL protocol for communication between the nodes as it provides a better utilization of network links while also providing least-cost pair-wise data forwarding. As hardware packet processing accelerator, we use the MPPA smart NIC from Kalray because of its parallel processing capability and ease of programming. In fact, this card supports the ODP API which is used for programming the networking data plane. It also allows reusing the code on any other platform that supports the ODP API.

The objective of our second contribution is to overcome the communication bottleneck of TRILL data plane implementation in the Linux Bridge Module and to do the performance evaluation of the newly designed solution. It actually occurs on each TRILL frame hop between different RBridges which requires the Outer Ethernet header that has been added to the original frame to be stripped off. Additionally, the hop count field of the TRILL header gets decreased and the frame gets encapsulated

into the new Outer Ethernet header (with the appropriate MAC addresses, after having consulted the forwarding table). In order to solve this problem, the idea is to offload the TRILL frame processing from CPUs on the MPPA smart NIC cards. The implementation is done by using the ODP API which permits building protocols from scratch and manipulation of header fields from various networking protocols. This allowed processing frames in parallel among all the cores that are available on the MPPA. The performance evaluation shows that it is possible to perform TRILL frame processing at full-duplex line rate (up to 40Gbps) for various frame sizes while also reducing latency when compared with the existing solution.

Furthermore, as our third contribution, we have investigated various network topologies and we compared their loads induced by the control plane traffic. We have provided an analytic model of the impact of different network topologies, that are well suited for the usage of smart NIC cards, on the load induced by the control plane. We take into account that Kalray MPPA cards have two 40Gbps which can be split with cables that support 40G QSFP+ to 4X10G SFP+ conversion. This allows connecting 8x10Gbps cables on one MPPA smart NIC. The goal is to build a mesh fabric network that does not use switches and that directly interconnects smart NICs. We have focused specifically on topologies used in PoP (Point of Presence) data centers that have a limited number of servers (30-80 servers) which is the case of a public platform of Gandi SAS. We examine how many TRILL Hello and LSP frames are generated per minute in various topologies but also the average distances between the RBridges. We have shown that *hypercube* topology is the most suitable for our PoP data center use case because it does not have a high control plane load and it has a better resilience than *fat-tree* while having a shorter average distance between the nodes.

1.4 Plan of the thesis

The rest of this thesis can be outlined as follows. In chapter 2 we start our state of the art by presenting solutions and protocols that are used for fabric networks. Then, we describe the terminology that is commonly used for this topic and we provide background information on packet processing on server-class network hosts. Additionally, we describe all the existing software-based and hardware-based solutions that aim to accelerate packet processing speed. Furthermore, we discuss their integration possibilities in virtualized environments and we compare them based on multiple criteria. Chapter 3 discusses our first contribution which is the architecture of the proposed solution. We compare the existing solutions that use specific hardware for packet processing. Then we gradually describe the advantages of the components that we chose to use such as Kalray MPPA smart NIC and the ODP API programming model. In chapter 4 we present our second contribution that is related to data plane offloading on a high-speed parallel processing architecture where we evaluate the network performance of the TRILL implementation on MPPA smart NIC. Afterward, in chapter 5 we present our third contribution in the form of a mathematical analysis of the impact of different network topologies on the control plane's load. Finally, we conclude in chapter 6 and we discuss about our short-term and long-term future works .

Chapter 2

State of the art

Summary

2.1 Introductory remark

The majority of the content in this chapter corresponds to the subject matter of our survey paper published in [56]. The goal of this chapter is to provide the state of the art on the existing packet processing implementations that are based either on software or hardware and that tend to accelerate packet processing speed. This is a very important part of our fabric network architecture that we try to build. The idea is to replace the classical three-layer network topology with the full-mesh network of links which is often referred to as the "network fabric". The advantage of such a solution is that it provides a predictable network performance with direct connectivity between any pair of nodes. Fabric networks have a flat architecture that is resilient, scalable and that decreases the number of hops needed for communication inside a data center. Packet processing speed is of crucial importance here as we plan to implement it directly on Linux servers either by using software acceleration techniques or hardware acceleration. For this reason, we survey various software-based and hardware-based implementations. Additionally, in order to have a better utilization of links and a network that can better suit the needs of a VDC, we analyze multiple fabric network solutions and protocols that are suitable for fabric network environments.

The rest of this chapter can be outlined as follows. In subsection 2.2 we present solutions and protocols that are used for fabric networks. Then, in subsection 2.3 we briefly present terminology that is commonly used for the fast packet processing 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. Subsection 2.4 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. Subsection 2.5 discusses different hardware implementations of packet processing. Afterward, in subsection 2.6, we present the integration possibilities in virtualized environments. Discussion on the latest approaches and future directions in packet processing is given in subsection 2.7 and we conclude in subsection 2.8.

2.2 Solutions and Protocols used for fabric networks

Recently, several proprietary data center networking solutions based on the fabric network concept have emerged and they aim to provide higher throughput across the network and to reduce latency. Such solutions replace traditional three-layer network topology with the full-mesh network of links which is often referred to as the "network fabric". Thus, any pair of nodes can have a predictable network performance with direct connectivity. Moreover, fabric networks have a flat architecture that is resilient, scalable and that decreases the number of hops needed for communication inside a data

center. When compared with the three-tier architecture, fabric networks reduce the amount of north-south traffic while the east-west traffic (which is typical for virtualized environments) encounters less network congestion. One of the main advantages is that VMs can use multiple paths to communicate with each other by using protocols such as TRILL or SPB (Shortest Path Bridging) instead of using a single fixed path calculated by the STP. Additionally, the links that are cut-off by the STP protocol will have a much higher utilization when multipathing is available.

2.2.1 Fabric network solutions

2.2.1.1 QFabric

Juniper Networks Quantum Fabric (QFabric) architecture delivers a data center design by creating a single tier network that operates like a single Ethernet Switch [57]. Such an architecture brings significant changes in scale, performance and simplicity while providing support for virtualized environments. Most of data center traffic today is server-to-server or east-west because of the adoption of service-oriented architecture (SOA)-based applications and virtualization [57]. In traditional three-tier architectures, packets need to transit multiple switches up and down the hierarchy in order to get to the destination. This can have a significant impact on transaction latency, workload mobility and real-time cloud computing interactions [57]. Additionally, operational expenses are higher in the traditional architectures because of the complex network management that needs to track the VM migrations and maintain connectivity between VMs in a tree structure. Juniper Networks QFabric creates a single-tier network that is operated and managed like a single, logical, distributed switch [57]. It consists of edge, interconnect and control devices that run Junos operating system.

In traditional data center networks, when the number of servers is increased, the new switches are added to the network in order to interconnect them. Each switch needs to be managed individually which is complicated in large data centers. Moreover, the amount of control traffic in the network also increases. In [57], the authors claim that in tree-based networks, 50% of network's ports (which are the most expensive ports) are used in order to interconnect switches and not to link servers, storage and other devices. Also, around 50% of network bandwidth is unavailable if STP is being run, as it disables almost half of the available links in order to avoid loops. With this in mind, capital and operational expenses are very high in traditional data center networks. The physical location of the server in a three-tier architecture also affects the application performance. A larger number of hops can increase latency which can further contribute to unpredictable application performance [57].

The inspiration for QFabric was drawn from switch design as inside every switch is a fabric which is a mesh that is completely flat [57]. It provides any-to-any connectivity between ports. A fabric network concept allows retaining simplicity by allowing multiple physical switches to be managed like and to behave as a single logical switch [57]. Any-to-any connectivity permits each device to be only one hop away from any other device. The QFabric architecture includes a distributed data plane that directly interconnects all ports one to another and an SOA-based control plane that provides high scalability, resiliency and a single point of management. Minimal administrative overhead is needed whenever a new compute node is added in a plug-and-play fashion. The three basic components

of a self-contained switch fabric - line cards, backplane and Routing Engines - are broken out into independent standalone devices - the QF/Node, the QF/Interconnect, and the QF/Director respectively [57]. QF/Node is a high density, fixed configuration 1 RU edge device which provides access into and out of the fabric [57]. QF/Interconnect connects all QF/Node edge devices in a full mesh topology. QF/Director integrates control and management services for the fabric and it builds a global view of the entire QFabric which could consist of thousands of server facing ports [57].

2.2.1.2 FabricPath

FabricPath [58] is an innovation in Cisco NX-OS Software that brings stability and scalability of routing to layer 2 [59]. Unlike in traditional data centers, the network does not need to be segmented anymore. FabricPath also provides a network-wide VM mobility with massive scalability. An entirely new layer 2 data plane is introduced and it is similar to a TRILL protocol. When a frame enters the fabric, it gets encapsulated with a new header that has a routable source and destination addresses. From there, the frame is routed and when it reaches the destination switch, it gets decapsulated into the original Ethernet format. Each switch in the FabricPath has a unique identifier called a Switch-ID which is assigned dynamically. Layer 2 forwarding tables on the FabricPath are built by associating the end hosts MAC addresses against the Switch-ID [58]. One of the advantages of FabricPath is that it preserves the plug-and-play benefits of the classical Ethernet as it requires minimal configuration [58]. FabricPath uses a single control protocol (IS-IS) for unicast forwarding, multicast forwarding and VLAN pruning [58]. It also allows the use of N-way multipathing, while providing high availability. Forwarding in FabricPath is always performed across the shortest path to the destination, unlike in layer 2 forwarding where STP does not necessarily calculate the shortest paths between two endpoints.

2.2.1.3 Brocade VCS Technology

Brocade defines the Ethernet fabrics in which the control and management planes are extracted from the physical switch into the fabric while the data plane remains on a switch level [60]. In this way, control and management become scalable distributed services that are integrated into the network instead of being integrated into the switch chassis. The advantage of such an approach is that the fabric scales automatically when another fabric-enabled switch is added [60]. In fact, a new switch automatically joins a logical chassis which is similar to adding a port card to a chassis switch [60]. The security and policy configuration parameters are automatically inherited by the new switch which simplifies monitoring, management and operations when compared with traditional layer 2 networks. Instead of performing configuration and management multiple times for each switch and each port, these functions are performed only once for the whole fabric. All switches in the fabric have the information about device connections to servers and storage which enables Automated Migration of Port Profiles (AMPP) within the fabric [60]. Furthermore, AMPP ensures that all network policies and security settings are maintained whenever a VM or server moves to another port of the fabric, without having to reconfigure the entire network [60]. Brocade VCS Technology implements all the properties and requirements of the Ethernet fabric and it removes the limitations of the classic Ethernet. For instance, it does not use STP but it rather uses the link-state routing. Moreover, equal-cost multipath forwarding

is implemented at layer 2 which brings a more efficient usage of the network links. VCS technology is divided in three parts: the VCS Ethernet that is used for the data plane, VCS Distributed Intelligence that is used for the control plane and the VCS Logical Chassis that is used for the management plane.

2.2.2 Communication protocols for fabric networks

2.2.2.1 TRILL

TRILL (TRansparent Interconnection of Lots of Links) [54] is the IETF's protocol that is used for communications inside a data center. The main goal of TRILL protocol is to overcome the limitations of conventional Ethernet networks by introducing certain advantages of network layer protocols. This protocol is considered to be a layer 2 and 1/2 protocol since it terminates traditional Ethernet clouds which is similar to what IP routers do. It has a simple configuration and deployment just like Ethernet and it is based on IS-IS protocol. TRILL provides least-cost pair-wise data forwarding without configuration, it supports multipathing for both unicast and multicast traffic and it provides safe forwarding during the periods of temporary loops. Additionally, TRILL provides a possibility to create a cloud of links that act as a single IP subnet from the IP nodes point of view. Also, TRILL is transparent to layer 3 protocols. A conversion from existing Ethernet deployments into a TRILL network can be done by replacing classical bridges with RBridges (Routing Bridges) which implement TRILL protocol.

RBridges perform the SPF (Shortest Path First) algorithm, thus providing optimal pair-wise forwarding. TRILL supports multipathing which provides better link utilization in the network by using multiple paths to transfer data between the two end hosts. This is enabled by the encapsulation of the traffic with an additional TRILL header. This header introduces the hop count field, which is a characteristic of layer 3 protocols, and allows for safe forwarding during periods of temporary loops. RBridges are also backward-compatible with IEEE 802.1 customer bridges [54]. More details about TRILL can be found in Appendix A.

2.2.2.2 SPB

SPB (Shortest Path Bridging) is the IEEE's protocol that is classified as 802.1aq [61]. It has been conceived as a replacement for older spanning tree protocols such as IEEE 802.1D that disabled all the links that did not belong to the spanning tree. Instead, it uses the shortest path trees (SPTs) which guarantee that traffic is always forwarded over the shortest path between two bridges. Due to a bidirectional property of SPTs, the forward and reverse traffic always take the same path. This also holds for unicast and multicast traffic between two bridges. SPB allows using multiple equal cost paths which brings better link utilization in the network. The control plane of the SPB protocol is based on the IS-IS (Intermediate System to Intermediate System) link-state routing protocol (ISIS-SPB) which brings new TLV extensions. This protocol is used to exchange the information between the bridges which is then used for SPT calculation. There are two variants of SPB. One uses the 802.1ad Q-in-Q datapath (shortest path bridging VID (SPBV)) and the other one uses the hierarchical 802.1ah MAC-in-MAC datapath (shortest path bridging MAC (SPBM)) [62]. Both of these variants have the same

control plane, algorithms and a common routing mechanism. SPB's control plane has a global view of a network topology which allows it to have fast restoration after a failure.

Both TRILL and SPB based Ethernet clouds scale much better than Ethernet networks that are based on spanning tree protocols [63]. One of the motivations for this thesis lays in the concept of fabric networks. However, the goal is to build a mesh network that is going to use highly performant smart NIC cards instead of switches that are interconnected just like the switches in the network fabric. Additionally, in our experimental evaluation we use the TRILL protocol for the communication between smart NICs which allows having higher utilization of links in the data center.

2.3 Fast packet processing

2.3.1 Terminology

There are two categories of procedures performed by the router: *time-critical* and *non-time critical* depending on how often these operations are performed [2]. *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. In order to reach gigabit data rates, these procedures need to be performed in an optimized way. On the other hand, *non-time critical* operations are referred to as *slow path*. Such operations are carried out mostly on packets which are used for maintenance, error handling and management. Implementation of *slow path* operations is done in a way that they do not interfere with *fast path* operations as the *fast path* always has a 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, in a broad sense, these two terms can also be 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.

2.3.1.1 Fast Path

There are two groups of *time-critical* operations [2]: *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 have a hardware implementation in high-performance routers. The reason behind this is that these operations need to be executed in real time for each individual packet. As it can be seen in Figure 2.1, 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.

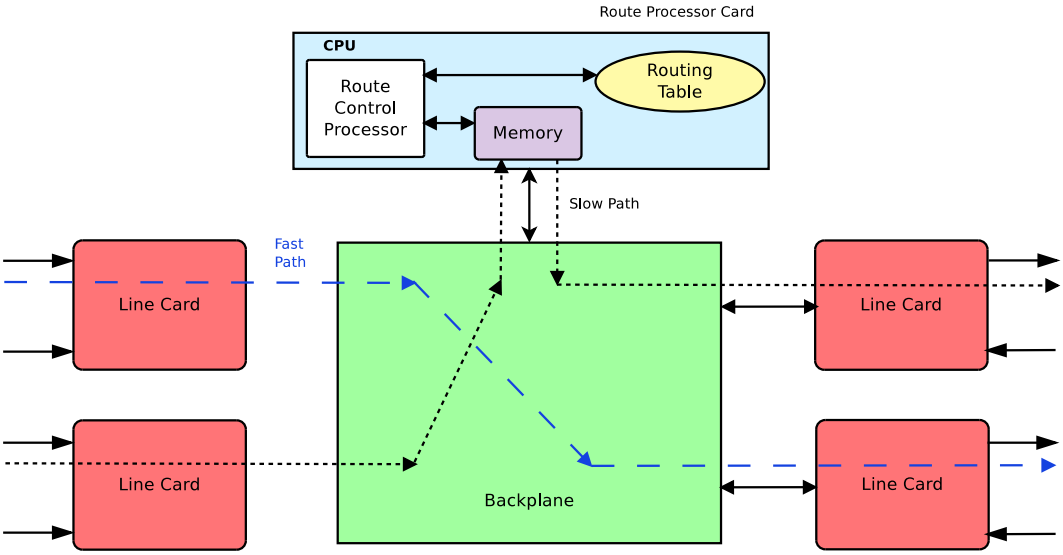


Figure 2.1: Components of a router. [2]

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 [2, 64]. 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.

2.3.1.2 Slow Path

Slow path packets are partially processed by the line card before getting sent to the CPU for further processing, as can be seen in Figure 2.1. In the *slow path*, there are *non-time-critical* operations which include [2]:

- 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

One of the *slow path* functions is the Address Resolution Protocol (ARP) processing 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.

Packet fragmentation and reassembly can be performed when a Maximum Transmission Unit (MTU) is not the same on different interfaces of a router [2]. 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. However, a downside of the fragmentation is the added complexity to the router's structure and reduction of data rate because an IP packet needs to be retransmitted each time a fragment is lost. Also, more headers are transferred for the same network throughput which consequently gives less transferred data. 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*.

2.3.2 Background on packet processing

Ever since the beginning of computer networking, 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 goal 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 [65]: 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 [66]: 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.

Multi-core architectures can improve the packet processing speed significantly if they are used in the right way. Nevertheless, partitioning the tasks across multiple concurrent execution threads is not enough because 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 [67]. For instance, each RX queue is assigned to a separate interrupt, and each interrupt is routed to a different core. This allows distributing the traffic from a single NIC across different cores. The number of queues is usually limited to the number of available processor cores [67]. Receive-Side Scaling (RSS) [68] 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 [69]. 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 [9]. 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.

2.4 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.

2.4.1 Click-based solutions

2.4.1.1 Click

Click [3] 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

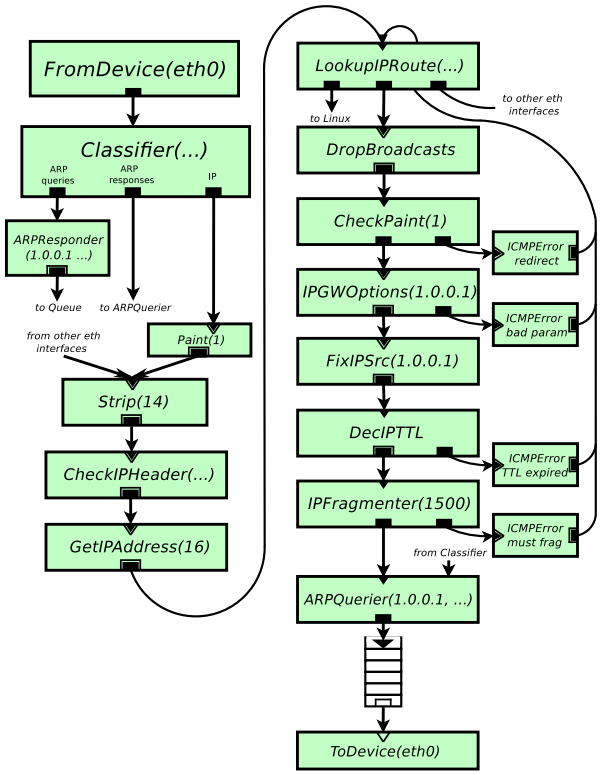


Figure 2.2: Example of a Click IP router configuration. [3]

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.

In Figure 2.2, 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 2.2 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 [3]. 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 [3]. 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 [3]. 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 [3]. 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 [3]. 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 [3]. 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* [3]. 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 [3]. 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, queue-

ing, 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.

2.4.1.2 RouteBricks

RouteBricks [31] 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 [31] relies on two key assumptions. The first assumption about RouteBricks router is that it can handle at least one router port of rate R [31]. 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 [31] 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 [70] 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 [71] 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 [31] 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 [31]. 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.

2.4.1.3 FastClick

FastClick [27] 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 [27]. 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 [72]. Also the CPU consumption

on per-packet memory management is eliminated [33]. 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.

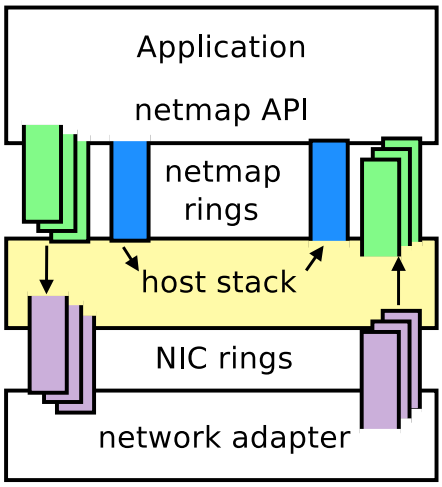


Figure 2.3: Netmap. [4]

2.4.2 Netmap

Netmap [4] is a framework which allows commodity hardware (without modifying applications or adding custom hardware) to handle millions of packets per second which go over 1..10 Gbit/s links. In [4] 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 [73].

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 [4], 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 2.3.

Netmap uses different techniques in order to attain its high performance [4]:

- 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;

2.4.3 NetSlices

NetSlice [5] 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. In [5], authors 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 2.4. 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

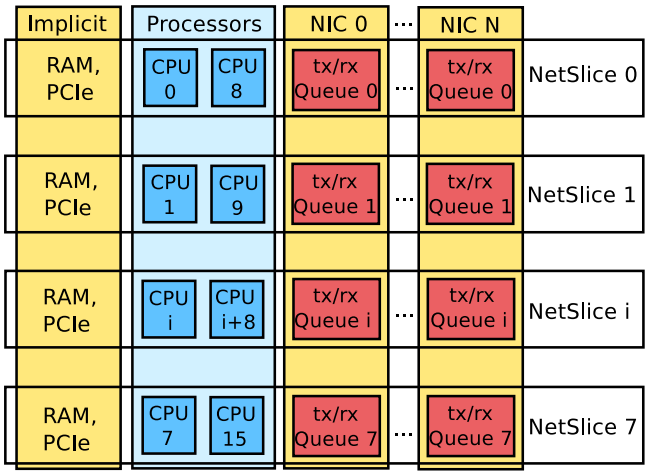


Figure 2.4: NetSlice spatial partitioning. [5]

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 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 [5], 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.

2.4.4 PF_RING (DNA)

PF_RING [6] 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 [74]. 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.

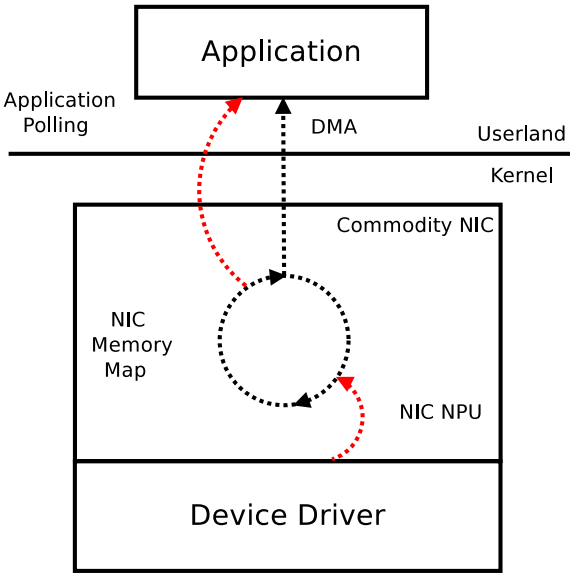


Figure 2.5: PF_RING with DNA driver. [6]

Interrupt livelock [75] 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.

In [74] the author claims 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) [67, 69] 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 2.5. 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 [6]. NAPI [76] 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 2.5.

2.4.5 DPDK

DPDK [28] 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 [7]. The 6WINDGate [77] 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) [28] 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 2.6.

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 [7].

Key software components of DPDK [7]:

- **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 [7].

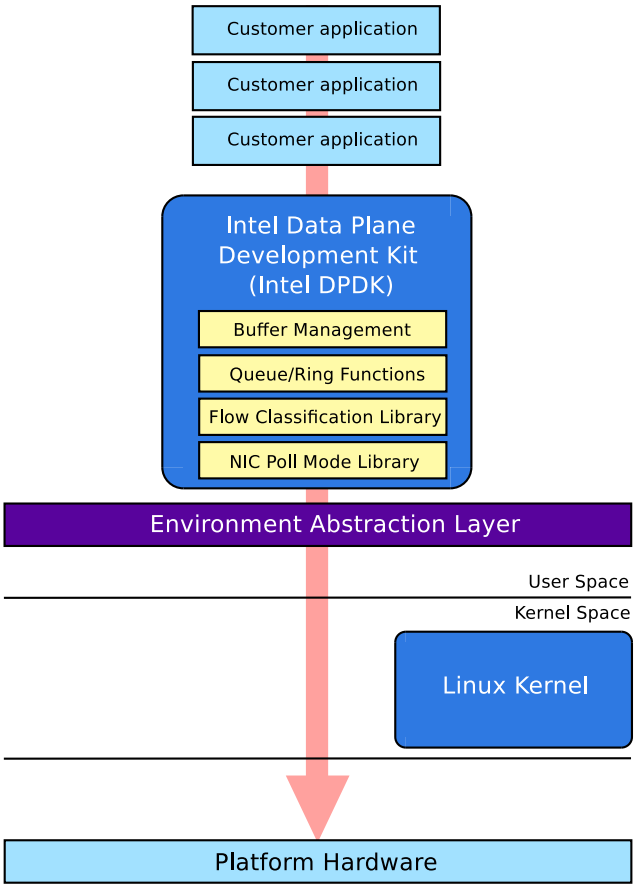


Figure 2.6: Major DPDK components. [7]

- **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.

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

2.5 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.

2.5.1 GPU-based solutions

2.5.1.1 Snap

Snap [32] 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 [78]. 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.

2.5.1.2 PacketShader

PacketShader [8] 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 [8] 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 [79].

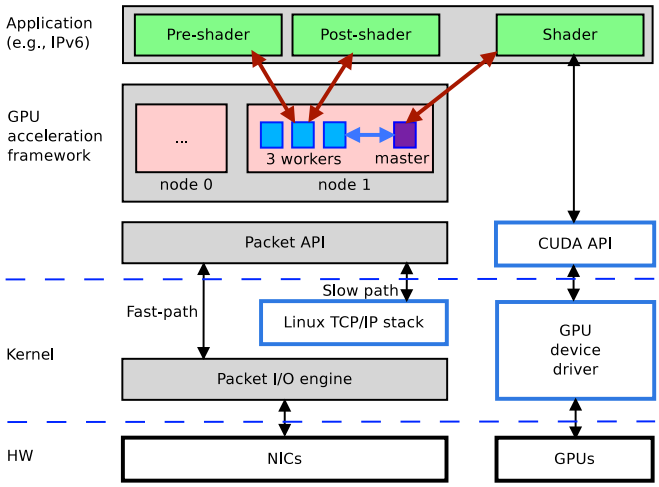


Figure 2.7: PacketShader software architecture. [8]

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 [8].

As a solution for the first problem *huge packet buffer* is introduced in [8]. 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.

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 [8] 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 [8]. 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 [8], 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 2.7 (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 [8]:

- **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.

2.5.1.3 APUNet

APUNet [9] 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 [80]. 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 [81]. 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 [9] 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

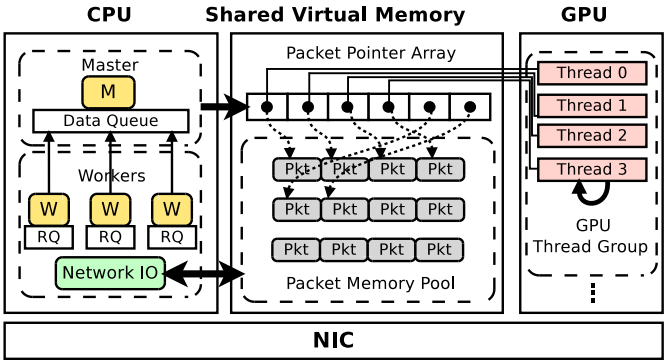


Figure 2.8: APUNet architecture: M (Master), W (Worker), RQ (Result Queue) [9]

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 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 [9]. 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 [9].

Just like PacketShader [8], the APUNet uses a single-master, multiple worker framework as shown in Figure 2.8. 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.

2.5.1.4 GASPP

GASPP [45] 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 [82]. 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 [82]:

- 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 [82]. 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 [82].

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 [82].

2.5.2 FPGA-based solutions

2.5.2.1 ClickNP

ClickNP [13] 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 [13]. 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 [13] 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 subsection 3.5.3.

2.5.2.2 GRIP

In [10] the authors 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 [10]. The

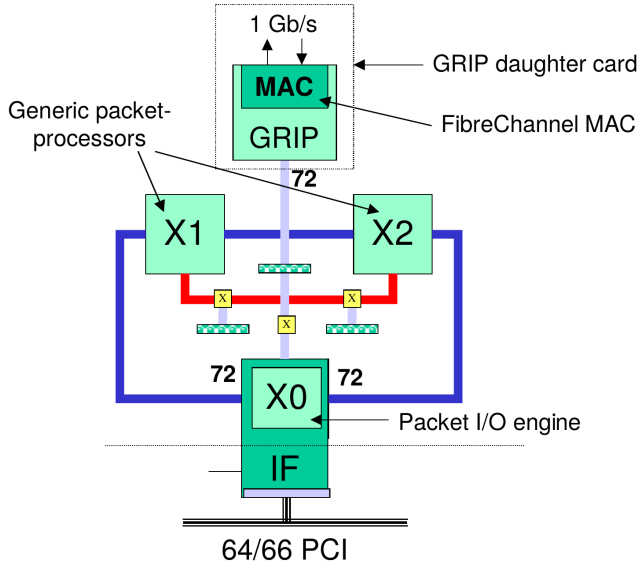


Figure 2.9: Block diagram of the SLAAC-1V architecture modified for GRIP. [10]

card is a SLAAC-1V based on the Xilinx Virtex architecture which provides three user-programmable Virtex FPGAs. In Figure 2.9 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 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.

2.5.2.3 SwitchBlade

SwitchBlade [11] 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 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 [11]:

- *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 [11].
- *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 [11]:

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.

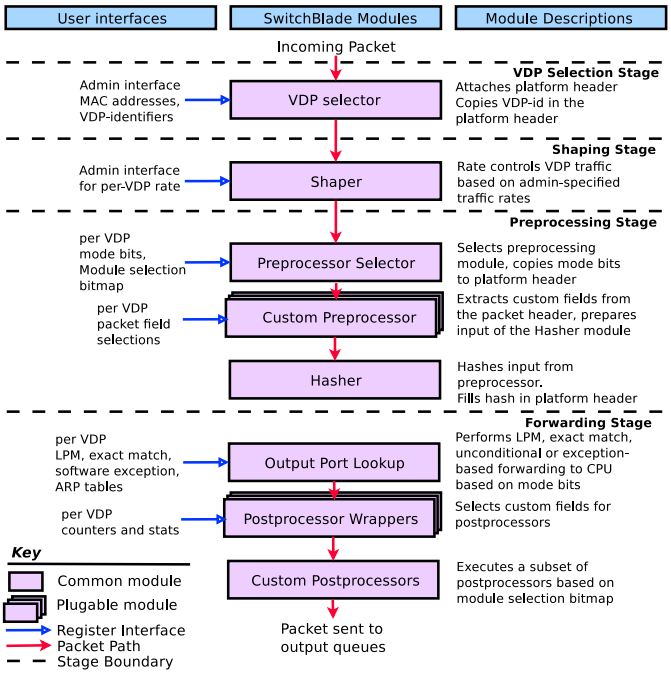


Figure 2.10: SwitchBlade Packet Processing Pipeline. [11]

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 2.10. 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.

2.5.2.4 Chimpp

Chimpp [41] 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 [41].

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 [41]. The main contributions of Chimpp are:

- generation of a top-level Verilog file from a high-level script [41]
- 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 [41]. 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.

2.5.3 Performance comparison of different IO frameworks

In [83] 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.

In [4] the Author 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 [83].

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 [83].

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 [84]. For this reason PacketDirect (PD) has been introduced as it boosts the current NDIS model with an accelerated network IO path [84]. 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

[84]. 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.

2.5.4 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 [80].

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 [85]. For this purpose, Intel has introduced a new CRC32 instruction [86].

2.6 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 [87], RouteBricks [88], FastClick [89], Snap [90], Netmap [91], NetSlice [92], PF_RING [93] and DPDK [94]. All these solutions codes are open source and fully available. Whereas, PacketShader code is partially available [95] according to their website [96]. 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.

2.6.1 Packet processing in virtualized environments

In [97, 98] Xen hypervisor [99] 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 [100] 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 [47] 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 [101] and KVM [102] hypervisors.

The same authors continued working on different modifications which tend to improve the packet I/O speed in VMs [48]. 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 [48]. 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 [48].

The ptnetmap [49] 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. [49]. 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 [49]. Implementation of ptnetmap

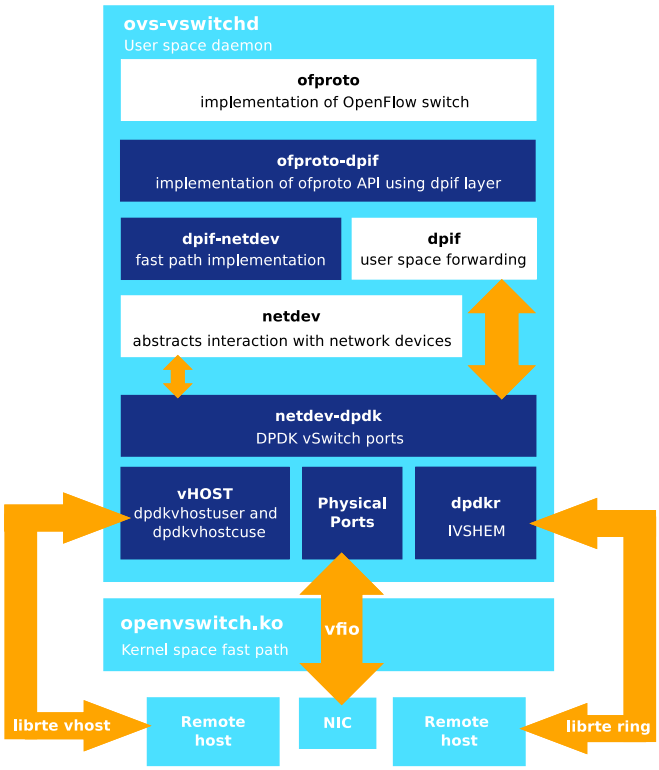


Figure 2.11: Open vSwitch with Data Plane Development Kit (DPDK) software architecture [12]

requires some extensions to netmap code along with minor changes in the guest device driver and the hypervisor. The successor of ptnetmap is ptnet [103] 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.) [103]. The ptnet overcomes these disadvantages and the authors claim that its performance is comparable with a widely adopted standard VirtIO [104].

NetVM [50] 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 [50]. However, in [49] the authors 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 [50].

An open source virtual switch with high performance is OvS-DPDK [12] which combines the functionalities of Open vSwitch [51] 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 [12]. 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 2.11, 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 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 the authors of [105] 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 [106]. 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 [106]. In [105], the authors claim that their OvS-DPDK architecture which uses integrated GPUs improves the throughput by 3x when compared to the CPU-only OvS-DPDK implementation.

In [107] the authors 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.

2.6.2 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 [87], RouteBricks [88], netmap [91], NetSlice [92], PF_RING [93], DPDK [94], FastClick [89]) 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 [3] 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 [108] 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 (2.2.16 in [108]) 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 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 [76] 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:

Table 2.1: 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 ([109],[110],[111]) - 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

- 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

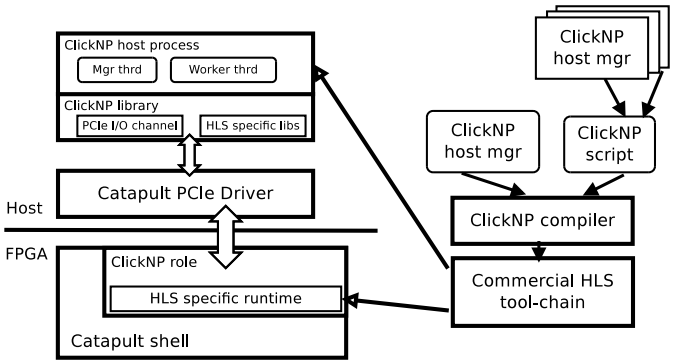


Figure 2.12: ClickNP architecture. [13]

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 [112]. 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 [113].

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 [45]. 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 [114] 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). In [13], the authors give a list of three HLS tool-chains compatible with ClickNP which are: 1) Altera SDK for OpenCL [109], 2) SDAccel Development Environment [110], and 3) Vivado Design Suite [111]. 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 2.12 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 [115] Gigabit Ethernet card driver, which also uses the XMACII [116] 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 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 2.1 summarizes all the constraints for all the solutions.

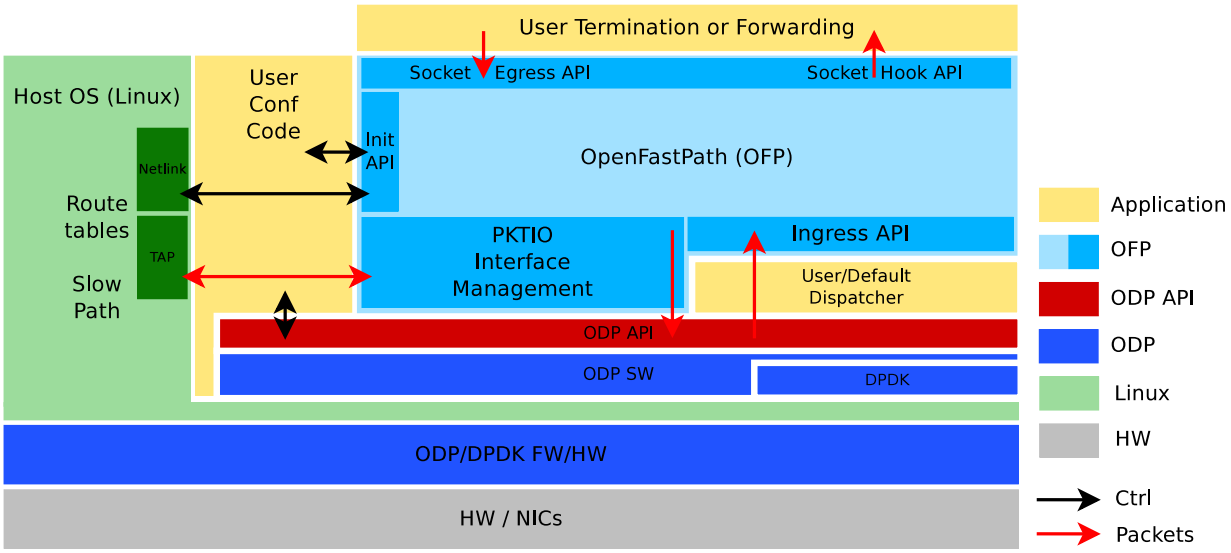


Figure 2.13: OpenFastPath system view. [14]

2.7 Latest approaches and future directions in packet processing

A novel Linux Foundation [117] 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 [118, 119]. 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 [120]. The VPP platform is a framework which can be extended and provides out-of-the-box production quality switch/router functionality [121]. 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. 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 [122]. ODP[123] is a set of APIs for the networking software defined data plane which are open-source and cross-platform. ODP consists of three parts [17]:

- 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 [17]. 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 [124, 125].

Another project intended for packet processing at high-speed is OpenFastPath (OFP) [126]. 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 [14]. OFP system view is shown in Figure 2.13. 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 [14]. DPDK is supported as well through the ODP-DPDK layer.

P4 [127] 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 [127]. 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 [128], P4GPU [129], P4-to-VHDL [130] and DC.p4 [131]. Additionally, there is a prototype of the P4 compiler [132] that generates C code directly from a P4 program that uses the Intel DPDK. A possible optimizer for the P4C compiler is presented in [133]. The authors of this work claim that their optimizer provides significant performance improvements.

OpenState [134], on the other hand, is an approach which allows performing stateful control func-

tionalities directly inside a switch without the need for the intervention of the external controller [135]. 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) [136] is concerned with building a programmable platform called SoftNIC [137] 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. The authors of [137] 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 [137].

2.8 Conclusion

In this section, we have presented three existing fabric network solutions: Juniper QFabric, Cisco FabricPath and Brocade VCS technology and we have explained the advantages that they bring when compared with traditional data center architectures. Also, we have presented TRILL and SPB protocols that are well-suited as a basis for a fabric network that we want to build.

Additionally, 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. We have also presented some hardware solutions that use FPGAs and GPUs. Moreover, we discussed the integration possibilities in virtualized environments of the described solutions, their constraints and their requirements.

All the analyzed packet processing 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.

Chapter 3

Fabric network architecture by using hardware acceleration cards

Summary

3.1 Introduction

An existing solution [1] is provisioned in the public platform of Gandi SAS, where the networking data plane is implemented in a modified Linux Bridge Module (open-source code available in [55]) as shown in Figure 1.3.a). A full mesh non-blocking layer 2 network was built based on TRILL protocol which does not need to use switches. However, this solution has a limited network performance. For this reason, our idea is to create a fabric network that is highly performant and that offloads packet processing tasks on hardware acceleration cards instead of performing them in Linux kernel on servers' CPUs. Such an architecture could reduce network latency and provide higher throughput. The main principle of a fabric network is to replace classical three-layer network topology with the mesh network of links. Its flat architecture is resilient because of a large number of links that connect any pair of nodes in the network. Since most of the nodes are directly connected between each other, a number of hops needed for communication inside a data center is decreased when compared with the three-layer network topologies. On top of that, using smart NIC cards can reduce the cost of the equipment significantly when compared with the cost of proprietary fabric network solutions [138, 139, 140] or dedicated networking hardware such as switches and routers.

In the next subsection, we introduce the problems and limitations of classical layer 2 architectures. Then we introduce the concept of fabric networks and we explain the VDC's requirements that TRILL protocol is able to fulfill which make it a suitable option for our fabric network. Afterward, in order to build such a network, we first need to choose which hardware to use for packet processing. In the following subsections, we compare several existing solutions described in sections 2.4 and 2.5 and we discuss about FPGA-based, GPU-based and other hardware solutions. However, there is another type of hardware called the MPPA smart NIC from Kalray that may be used for packet processing and that, up to our knowledge, has not yet appeared in the research literature related to fast packet processing field. For this reason, we dedicate a separate subsection to describe the advantages of this platform, on which subsequently we implement the data plane of the TRILL protocol.

3.2 Problems and limitations of traditional layer 2 architectures

Ethernet technology is still widely used by cloud service providers in order to interconnect the servers inside a data center because of Ethernet's simple operation and configuration. Ethernet bridges automatically learn the MAC addresses of the hosts that are connected to it by associating, in the forwarding table, the receiving port of the bridge with the source MAC address of the frame that arrived on that port. However, if the forwarding table does not contain the destination MAC address of the received frame, then the frame will be broadcast over all the ports of the bridge except the one that received a frame. Additionally, Ethernet frames do not have a hop count field that could limit the number of hops of the frame inside a data center. The downside is that any loop in the network could

cause a massive replication of frames which could consequently severely impact the functionality of the data center. Such a problem can escalate up until the frame forwarding capacity of one of the bridges is reached and link failures start to occur. In order to cope with this limitation, the STP (Spanning Tree Protocol) has been introduced. It allows to build a loop-free logical topology in the form of a tree where each pair of nodes has a single communication path in the network. In the case of a small network, VM migrations are possible as unique MAC addresses persist after each VM's change of location.

Nevertheless, in cloud environments there is a scaling problem when tens of thousands of VMs are present in a single data center. In order to create forwarding tables, network bridges broadcast frames across the network and this flooding causes the overload of the bridge's forwarding tables. In fact the CAM (Content Addressable Memory) of the layer 2 bridge is very responsive but at the same very expensive. For this reason, it has a limited amount of memory and it would be too expensive to replace all the bridges inside a data center with the ones that have CAM memories large enough to hold tens of thousands of VM's MAC addresses or even more.

Another problem is that STP cuts off a significant number of links in the network while building the spanning tree. Consequently, a majority of traffic between the VMs is done over the same subset of links that belong to the spanning tree. Since most of the data center's traffic today is communicated inside a data center, such a limitation in the number of active links becomes unacceptable as the need for higher throughputs is increasing exponentially. For this reason, it is often required to have multiple active paths between any pair of bridges. In the large data center networks, it is hardly possible to create a topology that has only one layer 2 segment for the whole data center because a high number of nodes may create congestion in the network.

In traditional data center networks, the adopted standard architecture consists of connecting a set of physical servers that belong to the same rack on a ToR (Top of the Rack) switch. These switches can also be considered as edge or access switches as shown in Figure 1.1. These access switches are connected to aggregation switches which are located in the aggregation layer. This layer aggregates all the traffic from the ToR switches and it also brings redundant paths between ToR switches and the core layer. The core layer is placed above the aggregation layer and it consists of CRs (Core Routers) that connect the data center with the outside world and they operate at the layer 3 of the OSI reference model. In the virtualized data center, the VMs are hosted on the servers that are connected to ToR switches (AccS switches in Figure 1.1). In this type of architecture, whenever a VM is migrated from one physical server to another one that belongs to a different layer 2 segment, its IP address needs to be changed. Moreover, the STP eliminates a possibility to use multipath communication between the servers.

With the virtualization of data centers come new challenging requirements such as: any-to-any connectivity with full non-blocking fabric, dynamic scalability, fault tolerance, seamless VM mobility, simplified management and private network support. For this reason, the goal of this chapter is to propose an architecture that will overcome the previously described limitations. Our architecture is based on the concept of fabric networks and it uses a protocol that fulfills the mentioned requirements. Additionally, we implement this protocol on a hardware acceleration card as such solutions tend to reach the highest performance among all the packet processing solutions on server-class network hosts.

3.3 Fabric networks

In order to reduce latency and provide higher throughput, several proprietary data center networking solutions have emerged that are based on fabric network switches. The idea is to replace the classical three-layer network topology with the full-mesh network of links which is often referred to as the "network fabric". The advantage of such a solution is that it provides a predictable network performance with direct connectivity between any pair of nodes. Fabric networks have a flat architecture that is resilient, scalable and that decreases the number of hops needed for communication inside a data center. Flat network design reduces the amount of north-south traffic load when compared with the three-tier architecture as the communication between VMs that are hosted on different servers is done over direct connections. This also means that east-west traffic, peculiar to virtualized environments, will encounter less network congestion. Moreover, VMs can use multiple paths to communicate with each other over protocols such as TRILL (more information on TRILL protocol in given in Appendix A) instead of using a single fixed path. This also enables an increased utilization of the entire network because the STP (Spanning Tree Protocol) may create inefficient utilization of certain links in the network. Additionally, such a solution is very well suited for virtualized data centers because it makes the configuration much easier. In the network fabric, all the switches can be managed as a single entity or individually and network administration is simplified when compared with traditional data center networks. There is no need for manual reconfiguration when a VM is migrated and all the devices in the fabric are aware of each other.

One of the best-known solutions for fabric networks is the Juniper's QFabric technology [138]. A fabric is a set of devices that act together to behave as a single switch according to [138]. In this architecture, all the access layer devices are directly interconnected by using a very large scale fabric backplane (called the *Interconnect device* in QFabric terminology). From a high-level viewpoint, such a system can be considered as a single non-blocking, low-latency switch [138] that supports thousands of multi-Gigabit Ethernet ports. Moreover, QFabric provides an enhanced scalability as it provides support for thousands of servers while being able to manage the QFabric system as a single system. On the other hand, there is a solution called the Cisco FabricPath [139] that has a similar idea to QFabric and it is based on a TRILL standard [59]. Moreover, Brocade has its own VCS Fabric solution [140] that is based on the same flat architecture principle.

One of the motivations for this thesis lays in the concept of fabric networks. However, the goal is to build a mesh network that is going to use highly performant smart NIC cards instead of switches that are interconnected just like the switches in the network fabric. Additionally, we use the TRILL protocol for the communication between smart NICs which allows having higher utilization of links in the data center.

3.4 TRILL protocol for communication inside a data center

Nowadays, the data center infrastructure gets virtualized more often than not, which allows the creation of the VDCs (Virtualized Data Centers) where physical servers are divided into mutually

	Commutation (L2)	Routing (L3)	TRILL
Multipath	No	Yes	Yes
Convergence	Slow	Fast	Fast
Connectivity	Inflexible	Flexible	Flexible
Scale	Limited	Important	Important
Configuration	Minimal	Intense	Minimal
Plug & play	Yes	No	Yes
Discovery	Automatic	Configured	Automatic
Learning	Automatic	Configured	Automatic

Figure 3.1: Advantages of TRILL protocol [15].

isolated virtual instances. These instances can have different properties such as the OS (Operating System) that they use, the quantity of CPU and memory resources that they use, etc. An increasing number of such virtual instances needs to be hosted on a VDC [1] while fulfilling the requirements such as scalability, fault tolerance, any-to-any connectivity, seamless VM (Virtual Machine) mobility, private network support and simplified management. A protocol that is able to fulfill such requirements is needed. One such protocol is TRILL (TRansparent Interconnection of Lots of Links) [54] and it has been conceived by the IETF (Internet Engineering Task Force). A more detailed introduction to TRILL protocol can be found in Appendix A.

In existing ethernet deployments, classical bridge devices can be replaced with RBridge (Routing Bridge) devices as they are retro-compatible with the former. TRILL has a simple configuration and deployment like Ethernet while also bringing the advantages of layer 3 routing techniques to layer 2. Additionally, TRILL protocol is transparent to layer 3 protocols. RBridges (also known as routing bridges or TRILL switches) are the devices that implement the TRILL protocol. They perform the SPF (Shortest Path First) algorithm, thus providing optimal pair-wise forwarding. TRILL provides multi-pathing for both unicast and multicast traffic [54] which allows better link utilization in the network as multiple paths may be used to perform the communication between the two hosts. This is enabled by the encapsulation of the traffic with an additional TRILL header that includes the hop count field. This field is characteristic of layer 3 protocols and allows for safe forwarding during periods of temporary loops. From Figure 3.1 we can see that TRILL protocol has advantages of both layer 2 and layer 3 protocols.

The main competitor of TRILL protocol is the IEEE's SPB (Shortest Path Bridging) protocol classified as 802.1aq [61]. Both SPB and TRILL enable multipath routing inside a data center and they are based on IS-IS protocol. SPB uses an existing IS-IS with TLV extensions while TRILL defines a new type of IS-IS instance with new PDU types [141]. The first one uses MAC-in-MAC encapsulation while the second one uses TRILL header encapsulation. Comparison of SPB and TRILL protocol is given in [141] from where we can see that the majority of properties are the same or very similar for both of the protocols. The question about which one should be used has been a subject of the "Great debate" [142] from which one can not claim that one protocol is better than the other. Both protocols

have their own pros and cons. Our choice remained to be TRILL as we have already done certain extensions of the TRILL protocol that turned out to be useful for Gandi public cloud infrastructure. One of them was a VNT (Virtual Network over TRILL) solution that introduces a new VNI Tag (24-bit segment ID) transported in the TRILL frame, thus providing a large-scale LAN segment number of up to 16 million [1]. Additionally, this would allow us to make a transition from the existing TRILL solution in the Linux kernel to the fabric network by gradually introducing the hardware acceleration cards that perform TRILL packet processing until the whole network is covered by hardware accelerators. By doing this, we would be able to overcome the communication bottleneck that is encountered when performing packet processing inside a Linux kernel. However, we think that SPB could also be very much suitable for our fabric network and we consider the TRILL protocol implementation only as one possible use case in the architecture that we propose.

3.5 Comparison of software and hardware packet processing implementations

In section 2.4 we introduced 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. We started with the Click Modular Router along with Click-based solutions such as RouteBricks and FastClick. Additionally, we described Netmap, NetSlices, PF_RING and DPDK solutions. In the following subsections, we compare all these solutions based on different criteria. A summary of comparison results is presented in Table 3.1. In section 2.5 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 3.2.

3.5.1 Comparison of software solutions

In this section, we compare 7 different software implementations of fast packet processing 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 3.1.

3.5.1.1 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 [3]. 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 [143] 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 [5] 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 [74].

FastClick is a high-speed user-space packet processor which is backward compatible with vanilla Click elements [27].

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.

3.5.1.2 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 [5]. 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. Accordingly, DPDK also benefits from zero-copy technique.

3.5.1.3 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 [8]. However, in [144] the authors 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 [31]. 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. PF_RING does not use batch packet processing whereas DPDK uses batch packet processing.

Table 3.1: 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

3.5.1.4 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 [67].

FastClick and DPDK also exploit parallelism.

3.5.2 Comparison of hardware solutions

3.5.2.1 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.

3.5.2.2 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.

3.5.2.3 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 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.

3.5.2.4 Operations in the user-space and kernel-space

PacketShader operates in user-space in order to take advantage of user-level programming [8]: 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 [145] 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 netmapcap). 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 [145].

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 user's register interface only [11]. When the register file is included, the programmer can write a user-space program by reading and writing to the register 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.

3.5.2.5 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 [8]. 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 [32] 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 NIC's DMA packet buffer.

3.5.2.6 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 [8].

The authors of Snap [32] have shown 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.

3.5.2.7 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.

Table 3.2: 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

3.5.3 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. [146]. 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 [147].

According to [80], 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 [106] 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 [148]. 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 [80]. 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 [149]. 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 the authors of [149] 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 [150] 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 [151]. 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 [152] 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 [153] 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 [154] with a reduction in average packet delay of 81.2%.

3.5.4 Discussion on FPGA-based solutions

In a setup used in [155] it has been shown that FPGAs are more energy efficient when compared to GPUs. In [156] the authors 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 [13]. 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 [157]. 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 [158] 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 [159, 160] 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 [161, 162] 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 [163] 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 [164]. 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 [164]. 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 [164]. The C-GEP solution relies on the use of high capacity Virtex 6 FPGA chip.

Figure 3.2: MPPA PCIe card [16]

3.5.5 Other hardware solutions

Today, one of the most common packet processing hardware offload applications consists of using NPs as accelerators. In the Ericsson whitepaper [165] 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 [165].

Knapp [166] is a packet processing offloading framework for Intel's Xeon Phi co-processor [167]. Xeon Phi co-processors assist the main CPU by adding instruction throughput and high local memory access bandwidth to the system [166]. 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 platform that uses specialized hardware is called MPPA (Massively Parallel Processor Array). We provide a detailed description of this smart NIC card in the following subsection.

3.6 Kalray MPPA processor

The MPPA2-256 (Bostan) processor was conceived by Kalray, a French company. It is the second processor of the MPPA (Massively Parallel Processor Array) processor family [16]. These processors are well-suited for networking, storage, compute-intensive and high performance embedded applications. There is a rising number of applications that move to the cloud and, as a result, there is an ever increasing amount of data traffic that needs to be handled in the data centers. Consequently, there is a growing need to offload intensive and real-time processing from mainstream processors to all kinds of programmable hardware. In this thesis, one of the contributions was to offload the data plane packet processing on the programmable smart NIC card and an example of the MPPA PCIe card that was used in our experimental testing is shown in Figure 3.2.

Generally speaking, MPPA is an alternative to FPGAs and GPUs. However, it has multiple advantages of both CPUs and DSPs. For instance, it has a CPU's ease of development as it is programmable in C/C++ programming language which provides it with 2-4 times faster time to market than FPGAs and ASICs. It also has DSP's energy efficiency and timing predictability. Additionally, MPPA has high-performance low-latency I/O just like FPGAs. Another advantage of MPPA processors is that they can be tiled together and operate simultaneously on massive calculations. Kalray also provides a simulator in the form of a VM which allows to develop, build and run applications on the MPPA processor simulator (without a need for a real processor). Debugging using the Kalray GDB debugger and trace tools are also available for this execution mode. Different levels of simulations are provided like single processor, single cluster, I/O cluster etc.

3.6.1 MPPA architecture

This smart NIC card has 256 user-programmable cores, while also having 32 system cores. In total 288 cores are integrated on a single chip and can run in parallel with high-performance and low power consumption. The group of 256 cores is divided into 16 clusters of 16 cores as shown in Figure 3.3. Additionally, each cluster has one system core which is also called a Resource Manager (RM), and it schedules the tasks on the user-programmable cores (called the Processing Engines (PEs)). Clusters and cores are represented by gray and green squares, respectively.

A single cluster is an elementary processing unit of MPPA architecture and it runs a specialized, low footprint OS, called NodeOS. Its architecture is shown on the right side of the Figure 3.3 while the global architecture of MPPA processor with all the clusters is shown on the left side. Each individual cluster has 2MB of shared memory that is used by the cores of that cluster. This memory is represented as a red box and it is used both by the RM and the PEs. Each 64-bit VLIW (Very Long Instruction Word) core (whether it is an RM or a PE) is characterized by its Kalray Bostan Instruction Set Architecture (ISA). The cores operate between 400MHz and 600MHz and the typical power consumption is between 10W and 20W.

On the left side of the Figure 3.3, the I/O subsystems are depicted as orange shapes in the upper-right and lower-left corner around the clusters. Each I/O subsystem has quad-core CPUs and a 40 GbE port (which also supports cables that provide a 40G QSFP+ to 4x10G SFP+ conversion if needed).

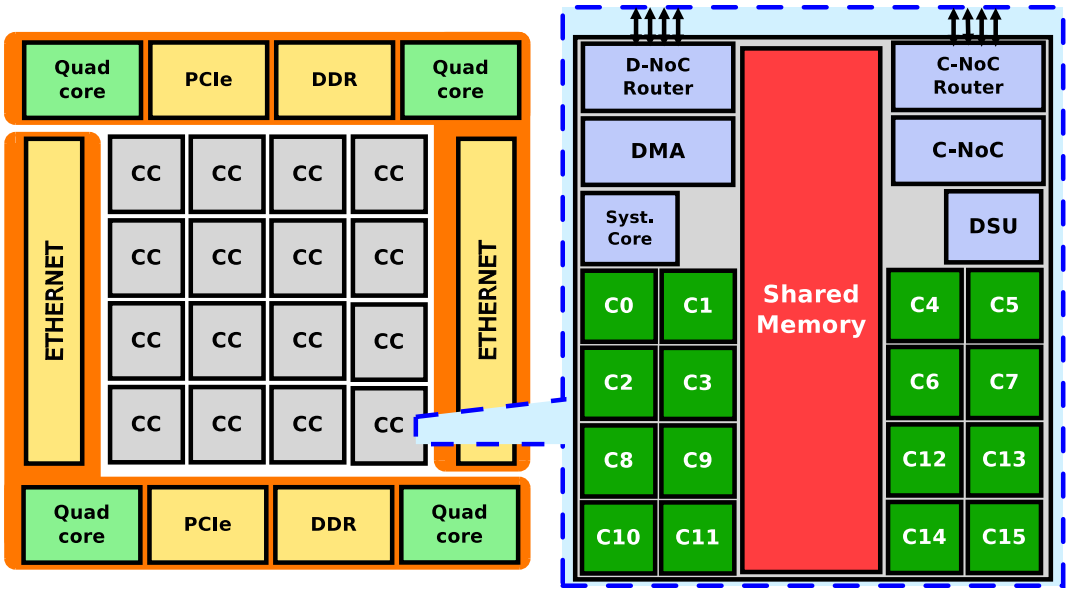


Figure 3.3: MPPA architecture [16]

Communication between I/O subsystems and clusters, but also between clusters themselves is provided by the use of NoCs (Networks on Chips). Each cluster has its own NoC while each I/O subsystem has multiple NoCs. This is shown in Figure 3.4 and the black curves represent the interconnection links between NoCs. In fact, there are two parallel NoCs with bidirectional links, one for data (D-NoC) and another one for control (C-NoC). The two NoC networks have a 2D torus topology [16] but in Figure 3.4 we show only one 2D torus topology in order to make it more clear. The D-NoC performs Remote DMA transfers of high bandwidth between the 16 clusters and the I/O subsystems. The C-NoC provides low latency messaging and synchronization between the clusters. The DMA (Direct Memory Access) unit of each cluster is used for communication between shared memory and the NoC.

Another important part shown in Figure 3.4 is the Smart Dispatch unit which is a hashing function implemented in hardware that is used for dispatching packets among different cores for their processing. It allows a developer to define a policy and the exact fileds of the packet's header on which this hashing function needs to be executed. More details about the usage of the Smart Dispatch unit is given in Subsection 4.2.3.

3.6.2 MPPA AccessCore SDK

Kalray MPPA technology includes the MPPA processor with the MPPA AccessCore. The later one represents a complete SDK (Software Development Kit) that integrates the reference standard GNU tools (gcc, g++, gdb, Eclipse). It also provides the debug and system trace tools for advanced optimizations. MPPA AccessCore includes explicit and implicit programming models. The explicit ones are C low-level/Lib and POSIX-level programming, while the implicit ones are OpenCL and ODP (Open-

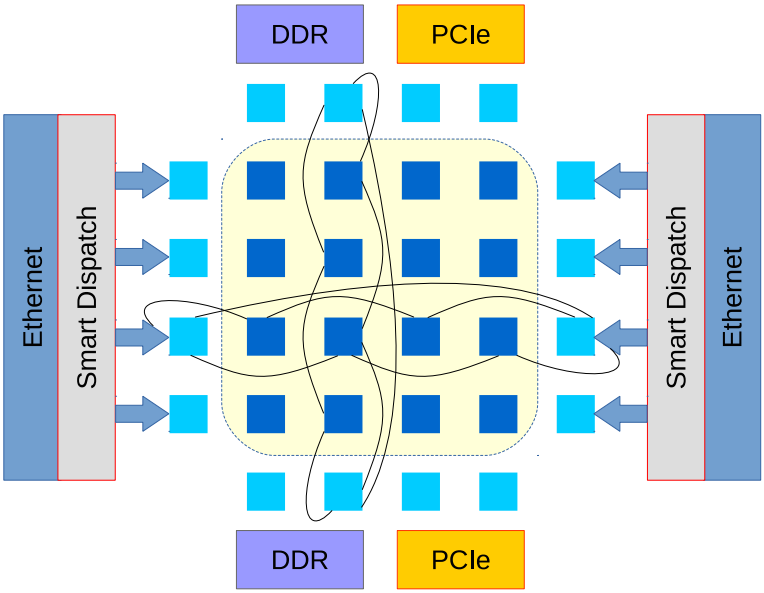


Figure 3.4: NoC interconnections and the Smart Dispatch unit.

DataPlane). Despite the fact that the implementation of TRILL protocol could be done in a low-level programming model which gives full control of all the aspects of the MPPA processor, we have decided to use the ODP API for our implementation. The main reason is the source code portability which allows us to reuse the same code on Linux ODP implementation or any other hardware accelerating card that supports the ODP API. For instance, there is a solution called ODP-DPDK [125] that puts ODP API on top of the DPDK solution in order to accelerate the packet processing performance of ODP API on Linux systems. In this way, we could theoretically build a network that uses different types of hardware that are used for the same purpose of packet processing. The more powerful hardware would be put in locations that have a greater need for processing resources like gateways, load balancers, etc. Additionally, ODP has been conceived specifically for the purpose of packet processing on any platform for which the ODP API is available. Therefore, the development process of TRILL protocol's data plane is much easier with ODP API when compared with other programming models supported by MPPA. Kalray's implementation of the ODP API is publicly available in [168]. More information on ODP API is given in section 3.7.

3.6.3 Reasons for choosing MPPA for packet processing

First of all, we have opted for a hardware acceleration of packet processing because hardware solutions achieve better performance in general. The main advantages of MPPA that make it a good choice for packet processing are the following:

- MPPA is programmable in C/C++ programming language
- high-performance and low-latency I/O just like FPGAs

- 2x40G QSFP+ interfaces
- support for ODP API which is very well suited for programming the networking data plane
- 2-4 times faster time to market than FPGAs and ASICs
- multiple MPPAs can be tiled together for even greater performance
- Smart Dispatching hardware unit for dispatching packets among clusters of cores

In chapter 4 we evaluate the network performance of the data plane TRILL implementation and it turns out that MPPA can significantly improve a network performance by reaching throughput of 40Gbps for certain packet sizes while reducing latency.

3.7 ODP (OpenDataPlane) API

ODP [123] is a cross-platform set of APIs used for programming the networking data plane. ODP API allows creating portable applications that can be executed on different types of platforms as shown in Figure 3.5. Each platform can transparently exploit its own acceleration properties based on the underlying hardware resources on which ODP is implemented. In fact, multiple existing ODP implementations are maintained independently and Kalray maintains the implementation of the ODP on their MPPA processor. ODP consists of three separate component parts that are related one to another. The first one is the ODP API specification that abstractly describes the functional model for data plane applications [17]. The goal of this specification is to provide the underlying platform with the ability to receive, process (manipulate) and transmit the packet data. However, there is no specific rule on how these processes need to be done and the exact implementation is left to the developer. For this purpose, the ODP API uses abstract data types whose implementation is a responsibility of ODP implementer for each specific platform. The ODP API has the following attributes [17]:

- open source, open contribution, BSD-3 licensed
- platform and vendor neutral
- application-oriented by covering functional needs of data plane applications
- specifies the functional behavior of ODP in order to ensure portability
- definition is given jointly and openly by platform implementers and application developers
- its architecture allows it to be efficiently implemented on a wide range of platforms
- maintained, sponsored and governed by the Linaro Networking Group (LNG)

ODP's second component is a set of ODP implementations where each one has been adapted for a specific target platform. The same functions on various platforms can be executed on CPUs or they can be offloaded to some hardware co-processing engine depending on the platform architecture and ODP API implementation. They can also be realized using specialized instructions that accelerate the functional behavior specified by the API [17]. Anyway, the functional behavior remains the same from

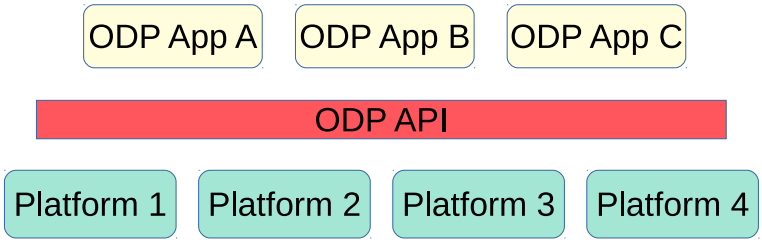


Figure 3.5: ODP application portability across platforms.

the application's point of view and it is independent from the actual realization on the platform. The main advantage of this approach is that anyone can create an ODP implementation that is specific to their platform. For instance, ODP can be directly deployed in the bare-metal environment or in the Linux user-space. Another advantage is that the owner can choose whether he prefers to release the open source code or keep it closed depending on business needs. Also, various implementations can have different release cycles but their compatibility still remains across different platforms.

The third component of ODP is its validation test suite that is used to ensure consistency between different ODP implementations [17]. In fact, it verifies if the functional behavior of a certain implementation correctly corresponds to the ODP API specification. This component makes sure that applications that use the ODP API can be executed on different platforms and that they remain portable. The tests can be performed both by users and platform providers in order to validate ODP implementations.

In general, ODP applications share the following characteristics [123]:

- their processing is divided into one or more threads that run in parallel
- their threads communicate between themselves and use synchronization mechanisms
- they receive(transmit) packets from(to) one or more packet I/O interfaces
- they examine, transform and perform different packet processing operations

In the case of ODP API implementation on the MPPA smart NIC, each cluster has its own ODP instance which uses cluster's shared memory and is independent of ODP instances from other clusters. However, they can all communicate by using libNOC library but also with two Ethernet interfaces on the MPPA machine.

3.7.1 ODP API concepts

The foundation of ODP API are the conceptual structures that facilitate the development of data plane packet processing applications. The main ODP concepts are [17]: Packet, Thread, Queue, PktIO, Pool, Shared Memory, Event, Buffer, Time, Timer and Synchronizer. In this subsection we provide a prompt description of the most important ODP concepts. Interested readers can refer to [17] for more details.

3.7.1.1 Packet

Packets are received and transmitted by the I/O interfaces. They are represented by handles of abstract type `odp_packet_t` that are specifically defined by each implementation. Packets are drawn from the pools of type `ODP_POOL_PACKET` and they have a rich set of semantics that allows to inspect them and manipulate them in complex ways [17]. Packets also support a rich set of metadata and user metadata where the later one allows any application to associate a specific side information for each packet that can be used by that application.

Packet objects are created upon their arrival at PktIO and they can be received directly by the application. Whenever a packet is sent, its object is freed automatically or it can be freed explicitly by using the `odp_packet_free()` function. ODP API also allows applications to create packets from scratch or make copies of the existing packets. The parse results are stored in the metadata section and various APIs permit applications to examine and/or modify this information [17].

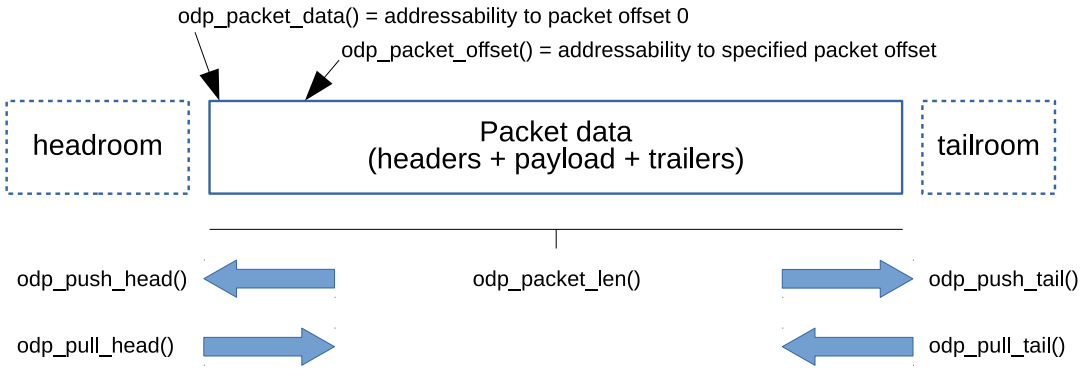


Figure 3.6: ODP packet structure [17]

In ODP API a *packet* represents a sequence of bytes that complies with a certain set of rules peculiar to the protocol in question. For instance, if Ethernet is used as a technology, then a structure of an Ethernet *packet* (or frame to be more precise) complies with the rules of the Ethernet standard. Each packet has its own *length* that represents a number of bytes that it contains and *offsets* that give references to different parts of a packet such as start of the layer 2 header, of the layer 3 header, of the packet data, etc. In fact, packet data consists of zero or more headers that are followed by zero or more *payload* bytes and also by zero or more trailers. Figure 3.6 shows different ODP functions that allow to examine and manipulate different parts of a packet. Packet size can be adjusted by manipulating a *headroom* and/or a *tailroom*. For instance, packets can be adjusted by *pulling* and/or *pushing* both of these areas.

3.7.1.2 Thread

ODP applications are organized into threads that perform the required work. They are divided into two groups: control threads and worker threads. Both of them are represented by the abstract

type `odp_thread_type_t` [17]. Control threads do not participate in the main packet flow through the system. Instead, they control, monitor and organize the operation of worker threads while also handling exceptions. Worker threads do the majority of packet processing work. They provide high packet rates and throughputs with low and predictable latency. More often than not, they operate on dedicated processing cores. Nevertheless, multiple threads can be pinned on a single core if needed, depending on the implementation platform (typically when a small number of cores is available).

Additionally, we would like to note that in ODP API implementation on MPPA platform, it is possible to define the number of threads that are going to be used specifically for packet reception. This value is contained in the abstract type `odp_platform_init_t` and it can severely impact the packet processing performance of this smart NIC.

3.7.1.3 Queue

A queue is a message passing channel that contains events [17]. Queues are represented by handles of abstract type `odp_pool_t`. Enqueue operations can add events to a queue while dequeue operations can remove events from a queue. Queues can be polled or scheduled. The former ones need to be polled explicitly in order to receive packets for instance, while for the later ones a scheduler selects and dispatches their events to requesting threads.

3.7.1.4 PktIO

Packet I/O (PktIO) is ODP's representation of I/O interfaces. It is a logical port that allows ODP processing of Ingress and Egress traffic. PktIOs are represented by handles of abstract type `odp_pktio_t`. For PktIO abstractions it is possible to set the MAC address, the number of queues, etc. This interface can be used directly in poll mode with adequate function that allow to send and receive packets over the specified PktIO.

3.7.1.5 Pool

Pools are represented by handles of abstract type `odp_pool_t` and they are shared memory areas from which elements may be drawn [17]. Pools are usually created during the application's initialization and they are destroyed during its termination. The two most relevant types of pools are *Packet* and *Buffer*. In fact, pools can be considered as memory areas to which packets are stored upon reception on PktIO.

3.8 Architecture of the fabric network by using the MPPA smart NICs

The architecture that we propose is based on the fabric network concept where we offload the TRILL data plane frame processing on the MPPA smart NICs instead of processing them in the Linux

kernel on servers' CPUs. The implementation is done by using the ODP API instances that are running on each MPPA cluster. This is shown in Figure 3.7. Such an architecture could provide higher throughput and reduce network latency. Additionally, a use of smart NIC cards can reduce the cost of the equipment significantly when compared with the cost of proprietary fabric network solutions. The existing implementation of the TRILL control plane is running in quagga daemon and it injects the calculated parts of the forwarding table into the shared memories of each cluster by communicating over the PCIe interface. VMs are hosted on the same Linux server that uses the MPPA as its networking interface. All the communication between the VMs inside a data center goes over the MPPA smart NICs. Data plane implementation of the TRILL protocol allows different RBridges to communicate inside a TRILL campus. We note here that VMs do not necessarily belong to the RBridge, but they are shown in the figure as they are hosted on the same server on which quagga daemon is running. Then, the different RBridges are connected between themselves inside a fabric network. A detailed discussion on the choice of the fabric network topology that interconnects the RBridges is given in Chapter 5.

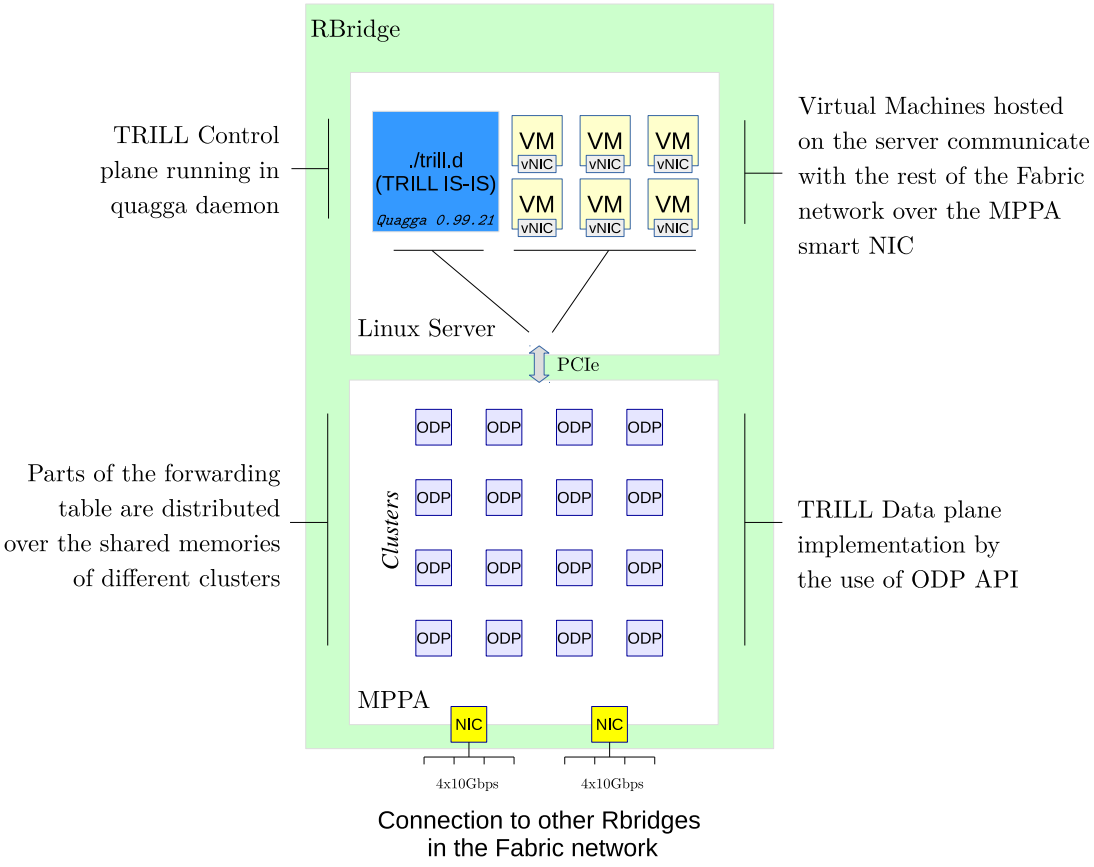


Figure 3.7: Architecture of an RBridge that uses the MPPA smart NIC inside a fabric network.

3.9 Conclusion

Cloud service providers and an increasing number of enterprises run their businesses by relying on data centers. The requirements of the Virtualized Data Center can not be fulfilled with classical layer 2 network solutions. For this reason, we propose an architecture that is based on a fabric network concept. IETF has proposed the TRILL protocol which is used for communication within the data center. We have chosen to use this protocol as a basis for our fabric network architecture as it turned out to have all the necessary advantages that allow an efficient communication between VMs in the data center. However, TRILL protocol is only one possible use case in our fabric network architecture and some other protocol like SPB could be used in its place. It is of vital importance to increase TRILL's packet processing performance on server-class network hosts as it can impose the limit to the functionality of the whole data center. For this reason, our idea is to offload its major part, which represents the communication bottleneck, on the programmable card which could significantly accelerate the TRILL frame processing and, consequently, the network throughput. In fact, packet processing speeds in software can barely support interface throughput which tends to increase nowadays, and for that reason many different packet processing solutions appeared lately. Combination of hardware and software tends to be the best option as hardware can alleviate the performance loss due to the virtualization of different parts of the system. Among the hardware accelerators, our decision is to use the MPPA smart NIC mostly because of its massive parallel processing power and ease of development. Additionally, among the programming models that it supports, we have chosen to use the ODP API because of the source code portability which allows us to reuse the same code on Linux ODP implementation or any other hardware accelerating card that supports the ODP API.

In the following chapter we deploy this offloading of packet processing on the MPPA smart NIC in order to quantify the performance increase in terms of throughput and delay.

Chapter 4

Data plane offloading on a high-speed parallel processing architecture

Summary

4.1 Introduction

In order to overcome a communication bottleneck of the existing solution [1] where the data plane of the TRILL protocol is implemented in a modified Linux Bridge Module, the idea is to offload the TRILL frame processing from servers' CPUs to MPPA (Massively Parallel Processor Array) cards

which can provide highly performant communication between the RBridges. In this chapter, we evaluate the performance of the TRILL's data plane implementation on the MPPA smart NIC from Kalray. Part of the content and the results obtained in this chapter have been published in [18].

4.2 System model and solution proposal

4.2.1 System model

The crucial question of gateway location where encapsulation/decapsulation of TRILL frames would take place has been a subject of discussion in one of the previous papers [1]. Apart from interconnecting TRILL equipment, one also needs to connect the standard Ethernet network with the TRILL network through a gateway. The TRILL gateway could be located in: 1. a silicon switch 2. the End Hosts in a software switch or 3. the Endpoints network interface. The second option is the most suitable one as the software implementation is done directly on a physical server, which is described in detail in [1]. This solution benefits from the following advantages:

- The TRILL encapsulated VM traffic is separated from server traffic which continues to commute in standard Ethernet format.
- VMs on the same host are exempted from encapsulating/decapsulating traffic between them.
- The number of MAC addresses that each server needs to handle is decent compared to the quantity that silicon switch would need to address
- Frame encapsulation overhead is reduced as frames are already generated on the server.
- Software implementation can easily be upgraded and does not require manufacturer support.

Despite the numerous advantages of TRILL implementation in the Linux kernel [55], we have identified a bottleneck which limits the network and server performance at the same time.

Actually, on each TRILL frame hop, there is a need to strip off the Outer Ethernet header that encapsulates the TRILL frame. Then, the TRILL header's hop count field needs to be decreased and the frame gets encapsulated inside a new Outer Ethernet header (with the assumption that the next hop link is of Ethernet type). The source MAC address of the Outer Ethernet header will be the MAC address of the encapsulating RBridge interface from which the frame is going to be forwarded. The destination MAC address of the Outer Ethernet header will be the MAC address of the next hop RBridge on the path to the egress RBridge. On the path of the frame to a destination, each RBridge consults its forwarding table upon frame reception in order to make the forwarding decision. The main problem here is that all these tasks need to be done for each TRILL frame and at every hop of the paths to the egress RBridges. Moreover, there is no fast path solution for the TRILL protocol. Therefore, each frame that is received by the RBridge network interface has to be managed by the OS and then has to be processed by the CPU. This represents the communication bottleneck as it will use up the CPU and memory resources [169]. Consequently, the lack of a fast path and the fact that the frame

processing is performed in the OS kernel are the reasons for the limited throughput of the TRILL frames.

TRILL implementation for the Linux Bridge module has been proposed in [1]. Nevertheless, the previously described throughput issue has been encountered. In order to solve this issue, in this section we propose the first - to the extent of our knowledge - TRILL fast path solution. For this purpose, we use a dedicated MPPA smart NIC from Kalray. On the MPPA card, we offload all the data plane processing of the TRILL frames. The goal is to improve the network performance significantly while preserving the server's resources.

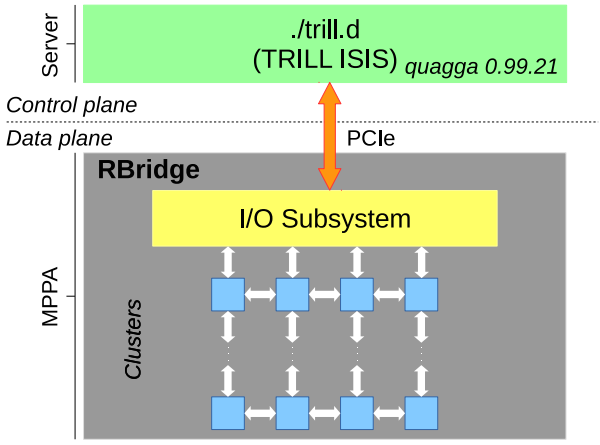


Figure 4.1: Communication between control plane and data plane [18].

The control plane of the proposed solution remains on the server while the data plane traffic is offloaded on the smart NIC. An implementation of the TRILL control plane has been done in the Quagga daemon [1]. It computes all the shortest paths between any pair of nodes and the distribution trees for multicast frames by using the information from LSP (Link State PDU) messages, exchanged between RBridges. Once the forwarding table is computed, it gets provided to the NIC in order to forward the TRILL frames. The PCI-Express bus and the NIC's I/O module allow transferring the forwarding table from the CPU to the NIC (Figure 4.1). Then the card dispatches parts of the forwarding table stored in the NIC's I/O module into the shared memory of each compute cluster (Figure 4.2). The goal is to dispatch in the shared memory of each cluster only the part of the forwarding table that will be used to forward the frames processed by that cluster.

As a matter of fact, the shared memory of each cluster is limited to 2 MB (Figure 3.3). Consequently, storing the entire forwarding table in the shared memory of a single cluster is not possible. To enable storing the entire forwarding table across all the clusters, we divide the forwarding table into 16 parts, one part per cluster. The division is done based on the egress nicknames such that each part contains the forwarding information for $2^{16}/16 = 4096$ egress nicknames. In subsection 4.2.3, we provide more information on the division of the forwarding table across different clusters.

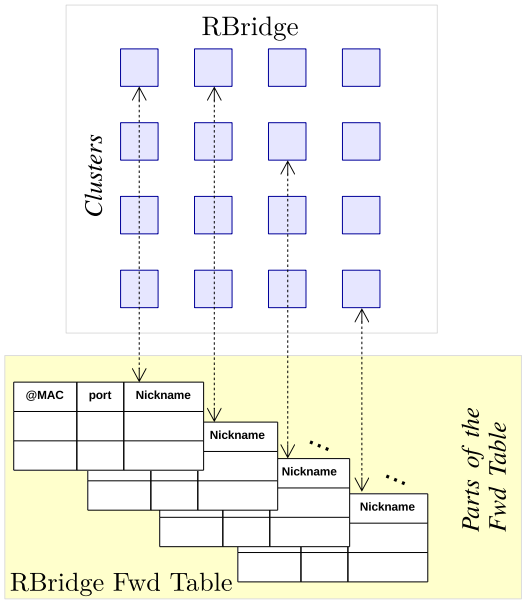


Figure 4.2: Partitioning of the forwarding table [18].

4.2.2 Frame journey

4.2.2.1 Control frame

Control frames are not processed by the smart NIC. Instead, they are received and sent directly to the CPU for processing. The implementation in Quagga follows the TRILL RFC [54]. However, in the following sections we analyze the load of the control plane depending on the topology used in the PoP data center.

4.2.2.2 Data frame

We have developed a TRILL fast path solution that offloads the data plane processing of the frames on the MPPA smart NIC in order to improve the throughput of the TRILL traffic. The MPPA card encapsulates the data packets from the VMs into the TRILL data frames. Moreover, it decapsulates the TRILL frames addressed to the host of the MPPA card. Additionally, this smart NIC also performs the forwarding of the TRILL frames in the TRILL network. For this purpose, the smart NIC uses a forwarding table that is computed by the control plane. MPPA is also in charge of updating the TRILL header by decrementing the hop count (TTL) field and by replacing the outer Ethernet header of the frame.

The smart NIC performs the steps shown in Algorithm 1 when sending a frame. In line 2 of the Algorithm 1, the MPPA checks if there is a destination RBrdige Nickname that is associated with the destination MAC address. Similarly, the Algorithm 2 details the steps performed by the smart NIC when receiving a frame.

Algorithm 1 Sending a frame

```
1: if (destNickname is known) then
2:    if (∃ destNickname ↔ destMAC) then
3:       MPPA encapsulates in a unicast TRILL frame;
4:       (with srcNickname ← MPPANickname)
5:       MPPA encapsulates in an Ethernet frame;
6:       (and sends towards a next hop)
7:    end if
8: else
9:    encapsulate in a multicast TRILL frame;
10:   (with srcNickname ← MPPANickname
11:   and destNickname ← Nickname of the distribution tree)
12:   encapsulate in an Ethernet frame;
13:   (and sends towards a next hop)
14: end if
```

Algorithm 2 Receiving a frame

```
1: if (destNickname = MPPANickname) then
2:    MPPA removes the TRILL header;
3:    lookup for destMAC in the local DB;
4:    if found then
5:       send frame to the host for delivery to the VM;
6:    else
7:       if no destination then
8:          drop the frame;
9:       end if
10:   end if
11: else
12:   MPPA removes the outer Ethernet header;
13:   decrease the hop count in the TRILL header;
14:   encapsulate in an Ethernet frame;
15:   (and sends towards a next hop)
16: end if
```

4.2.3 Implementation of TRILL data plane on the MPPA machine

We implemented the TRILL data plane on the MPPA machine by using the ODP API. When a packet arrives on one of the Ethernet interfaces of the MPPA, it gets dispatched by the Smart Dispatch unit on one of the active clusters. Inside MPPA, the NoCs are used for the interior packet routing and each cluster has its own NoC as shown in Figure 3.4. Interconnection links between NoCs are represented as black curves. The hardware packet dispatcher forwards each packet on one of the active

clusters depending on the user-defined policy (rule) that is given to the dispatcher. Active clusters are clusters which have an open ODP Packet I/O (PktIO) logical port that allows receiving (transmitting) packets from (to) the NIC. By default, the Smart Dispatch unit will forward the packets on the active clusters in RR (Round Robin) manner if no policy is set.

Smart Dispatch unit can perform several different policies. Selective Round Robin is one of the available policies and it allows a user to predetermine the cluster to which the frame is sent depending on the result of the Smart Dispatch hashing function. The Smart Dispatch unit performs a hashing function on user-defined bytes of the received frame's header. Our goal is to divide all the TRILL traffic in 16 equal parts by using the Smart Dispatcher in order to balance the computing load on the 16 processing clusters of the NIC. To achieve that, we apply the Smart Dispatcher hashing function on the 2-byte-long Egress RBridge Nickname field of the TRILL header. The NIC possesses 16 computing clusters while the number of possible nicknames is $2^{16} = 65536$. Since the hashing function always has the same number of possible outputs as the number of active clusters, we end up with 16 groups of nicknames (1 per cluster). Additionally, the dispatcher always tries to perform a uniform division of its hashed values, therefore each one of the 16 groups contains $2^{16}/16 = 4096$ nicknames.

Finally, as the Selective Round Robin policy allows the user to choose the cluster for each of the 16 resulting groups of the hashing function, we have as a result that each cluster needs to have only 1/16 part of the whole forwarding table (2^{16}/16=2^{12} entries) as it will always receive the addresses from the same group. This will not only reduce the amount of used shared memory inside a cluster (shown in Figure 3.3) which is already small (2MB), but it will also accelerate the lookup process in the forwarding table whose size per cluster is reduced significantly. The partitioning of the forwarding table is depicted in Figure 4.2. This dispatching procedure, along with the implementation of the steps described in Section 4.2.2.2 by using the ODP API, allows to efficiently perform the processing of TRILL frames on the MPPA processor.

4.3 Performance evaluation

4.3.1 Experimental setup and methodology

The existing control plane is already fully tested and running on more than 400 machines in public cloud infrastructure of Gandi SAS and the proof of concept with MPPA is sufficient to show that this card is able to execute highly performing data plane TRILL frame processing for the three different cases: encapsulation, forwarding and decapsulation of TRILL frames. Thus, for the performance evaluation of the proposed architecture, we used two servers which are directly connected over the optical 40Gbps links to two QSFP+ interfaces of the MPPA processor card. The traffic has been generated from two sides by both servers (two Supermicro SYS-5018R-MR) by using the Intel XL710 (40GbE QSFP+) NICs. The Pktgen-dpdk traffic generator (Pktgen version 3.4.9 with DPDK 17.08.1) was used to generate Ethernet and TRILL frames with various frame sizes. Then, both servers received the traffic processed (forwarded) by the MPPA machine. The servers run Ubuntu 16.04.3 LTS. Packet generator was able to saturate the 40Gbps network interface when generating Ethernet and TRILL frames for all different sizes except for the smallest ones (64B for Ethernet and 84B for TRILL frames,

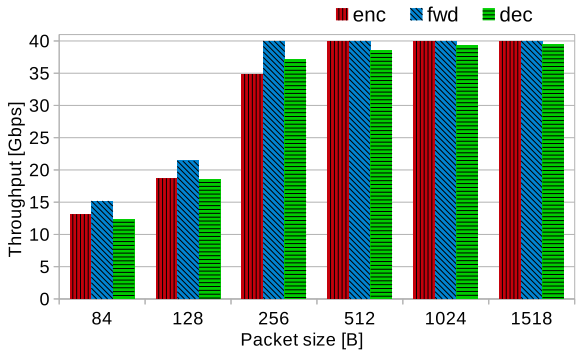


Figure 4.3: Throughput in the case of encapsulation, forwarding and decapsulation of TRILL frames.

respectively). This is due to the limited achievable packet rate of the Intel XL710 NIC for small packet sizes [170] (maximum packet rate is 42.56Mpps according to our measurements when generating traffic with Pktgen-dpdk).

For the round-trip time (RTT) measurements we used a single server that generated TRILL frames towards the MPPA processor. All the frames would be forwarded back to the same server in order to evade synchronization inaccuracies between different servers. We noted a timestamp for each packet both when transmitting and receiving them on the interface. Then, the mean round-trip time for different TRILL frame sizes was calculated based on this data.

4.3.2 Throughput, latency and packet processing rate

We have done the evaluation of the throughput that each server receives from the MPPA for three different cases: encapsulation of Ethernet frames in TRILL frames, forwarding of TRILL frames and decapsulation of TRILL frames to Ethernet frames. These three cases are shown as red, blue and green hatched bars in Figures 4.3 and 4.4, respectively. The numbers on the x-axis in all three cases represent the sizes of the frames with all three headers included (outer Ethernet, TRILL and inner Ethernet header).

Firstly, in the case of forwarding the size of the frame at the input is the same as at the output of MPPA for all frame sizes on the x-axis. As described in subsection 4.2.2, the frame has all three headers before and after the forwarding. From Figure 4.3 we can see that the forwarding throughput reaches the maximum values for packet sizes ranging from 256B to 1518B and that it is limited by the speed of the networking interface (40Gbps) and not by the processing power of MPPA. For packet sizes of 84B and 128B, a limitation is due to the maximum packet rate supported by MPPA which is around 18Mpps. For all the other frame sizes, the packet rates correspond to the maximum throughput achievable on the link (40Gbps).

In the case of encapsulation, the value on the x-axis represents the size of the encapsulated frame (with all three headers). This means that at the input of the MPPA the frame size is smaller for 20B (=14B + 6B) which represents the the sum of the Ethernet and TRILL header sizes. For instance, in

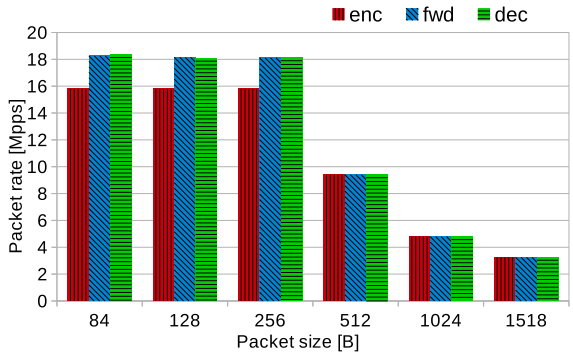


Figure 4.4: Packet rate in the case of encapsulation, forwarding and decapsulation of TRILL frames.

the case of the first bar (in Figures 4.3 and 4.4), the (inner) Ethernet frame of 64B is encapsulated in the 84B (=64B+20B) frame. From Figure 4.3 we can conclude that in the case of encapsulation the MPPA is able to encapsulate Ethernet frames with TRILL headers at full-duplex line-rate (40Gbps) for the frame sizes ranging from 512B to 1518B. However, even though at the output of the MPPA we have the maximum throughput (40Gbps), some of the packets that are sent for encapsulation are dropped. This happens because it is not possible to encapsulate all the packets of certain size if they are sent at NIC's maximum throughput to MPPA as they would require an even higher throughput at the output. For this reason, the encapsulation throughput is limited by the speed of the networking interface for the frame sizes ranging from 512B to 1518B. Other frame sizes (ranging from 84B to 256B when encapsulated) do not reach the line-rate throughput. Their data rate is limited due to the maximum packet rate supported by MPPA which is around 18Mpps as shown in Figure 4.4. For larger packet sizes, the packet rates correspond to the maximum throughput achievable on the link (40Gbps).

For the case of decapsulation, the value on the x-axis represents the size of the frame that gets decapsulated (with all three headers). This means that at the input of the MPPA the frame size is larger for 20B (=14B + 6B) than at the output of MPPA. For example, in the case of the last bar (1518B), the frame gets decapsulated to the frame of 1498B (=1518B-20B). From Figure 4.3 we can conclude that the decapsulation throughput does not reach the line rate even for large frame sizes which is expected since it is not possible to have the maximal throughput at the input and at the output in the case of decapsulation. However, from Figures 4.3 and 4.4 we can see that in the case of decapsulation packet rate maintains the maximum speed for frame sizes ranging from 256B to 1518B. This means that all the packets get decapsulated but the lower throughput (from Figure 4.3) corresponds to the smaller size of decapsulated packets. For frame sizes ranging from 84B to 128B, line-rate throughput is not reached, due to the limited packet rate supported by MPPA.

In Figure 4.4 we can see that the rate with which a server receives the processed frames in all three cases is not bounded by the processing capabilities of the MPPA processor the for the large frame sizes (512B or larger). In fact, for these sizes a packet rate reaches the top limit for the given NIC throughput of 40Gbps. Both the results of received throughput and packet rate show that offloading packet processing on parallel processing architecture can largely overcome the limitations of software packet processing. Additionally, in all three cases for large packet sizes, the performance was bounded

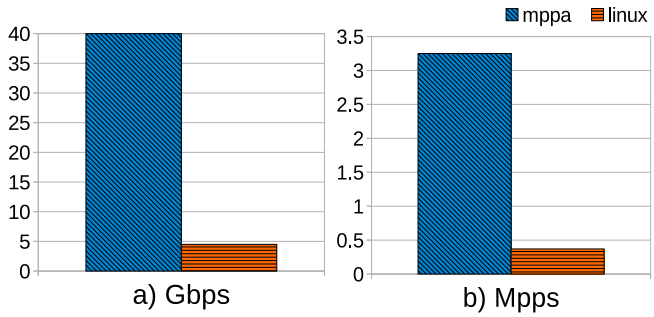
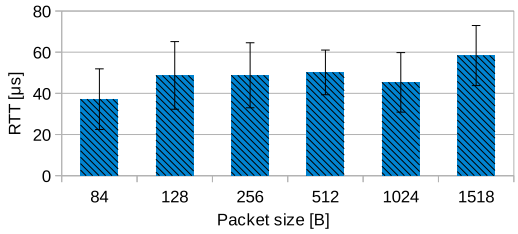


Figure 4.5: Performance comparison of MPPA and Linux solutions for TRILL frame forwarding (1500B): a) throughput and b) packet rate.



by the NIC throughput and not by the processing power of the MPPA card.

Figure 4.5 shows the TRILL frame forwarding performance comparison of the existing Linux solution and the MPPA solution that we propose. In the forwarding tests, the frame size has been set to 1500B as in [1]. On Figure 4.5.a) we can see that the throughput is more than 8 times higher when MPPA is used, compared to the existing TRILL frame forwarding solution for Linux. Accordingly, the packet rate on Figure 4.5.b) is significantly higher when MPPA is used. In the case of MPPA, the packet rate is limited by the NIC's throughput (40Gbps), as can be seen in Figure 4.4 where packet rates can go even up to 18Mpps (for smaller frame sizes) compared to less than 4Mpps in Figure 4.5 (for 1500B frames).

Moreover, the Round-trip time needed to send a TRILL frame to and receive it back from the MPPA processor is shown in Figure 4.6. The Cumulative Distribution Function (CDF) of RTT shows what is the probability that the RTT is lower than a certain value for different packet sizes. The RTT is significantly reduced when compared to the results that were obtained in the previous work [1], where TRILL frame processing was performed directly on the server. The main advantage when using the MPPA smart NIC is that upon arrival, the TRILL frame gets dispatched directly into the shared memory of the cluster responsible for processing it. This fast-path processing allows it to quickly process the frame and send it back to the network interface that needs to forward a packet.

4.4 Conclusion

In this chapter, we have presented an architecture that offloads the data plane packet processing on

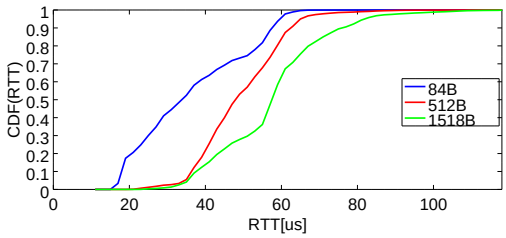


Figure 4.6: Round-trip time.

a programmable high-speed parallel processor. Despite the benefits of TRILL protocol implementation in the Linux Kernel, which is provisioned in the public cloud infrastructure of Gandi SAS, we found a packet processing bottleneck that limits performance. Packet processing speeds in software can hardly keep up with today's NICs throughput. For this reason, the hybrid combination of hardware and software turns out to have the highest performance. Our experimental results show that the MPPA processor can provide significant performance improvements in packet processing while exploiting parallelism of its manycore architecture. It can process the packets at full-duplex line rate (up to 40Gbps) for different packet sizes and reduce latency. Since MPPA can be integrated over the PCIe, this allows it to replace the server's NIC card and offload the computation and memory intensive workload of packet processing on a parallel processing architecture while preserving server's resources. This means that integrating the TRILL protocol in such devices would allow creating a fabric network without a need for a switch. Furthermore, the ODP API on the MPPA processor allows the easy integration of other networking protocols that could be suitable for a fabric network.

Chapter 5

Analysis of the fabric network's control plane for the PoP data centers use case

Summary

5.1 Data center network architectures

In this chapter, we present and discuss three different network topologies that are suitable for our PoP (Point of Presence) data center use case where a number of servers is limited. Each topology has different scaling properties when the number of servers is increased. Data center network architectures have been a subject of research for quite a while now. As stated in [19], typical network topologies

consist of two-level or three-level trees. The former ones have a core and an edge level, while the latter ones have an additional aggregate layer.

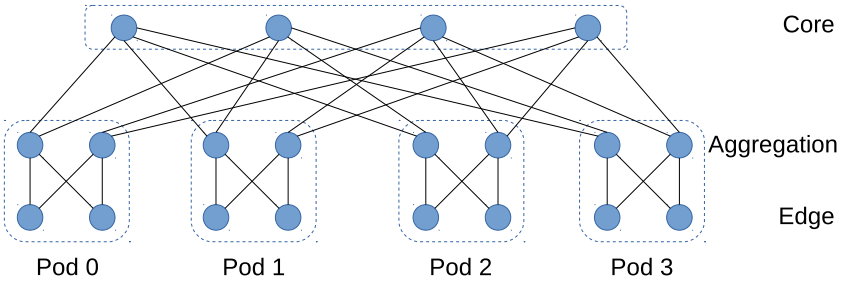


Figure 5.1: Fat-tree topology (k=4) [19].

Fat-tree topology, one of the best-known interconnection architectures, has been proposed by Leiserson [171]. Despite the fact that in chapters 2 and 3 we claimed that our goal is to build a fabric network with a flat architecture, in this chapter we analyze the behavior of the *fat-tree* topology as well in order to be able to compare it with the other two topologies. The authors of [19] proposed a special case of a Clos topology that they also called *fat-tree* topology. They formalized it as in Figure 5.1 where the network is divided into k pods where each one contains two layers of $k/2$ switches. The lower layer is the edge layer where each k-port switch connects with the aggregation layer over the $k/2$ ports. Each switch in the aggregation layer uses the $k/2$ ports to connect with the core layer and the remaining $k/2$ ports to connect with the edge layer. The core layer contains $(k/2)^2$ k-port switches [19].

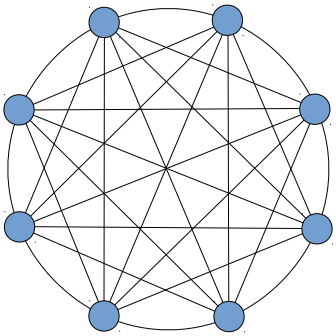


Figure 5.2: Full-mesh topology.

All the switches are connected directly to each other in the *full-mesh* topology, as shown in Figure 5.2. This allows any pair of switches to communicate independently from the state of the other switches in the network. The main advantage of this topology is its resilience as the number of redundant paths between any pair of nodes is maximized which means that link failures do not have a severe impact on connectivity between the nodes. The disadvantage of this approach is that many switch ports are used for the interconnection with other switches in the network and this number is linearly dependent on the size of the network.

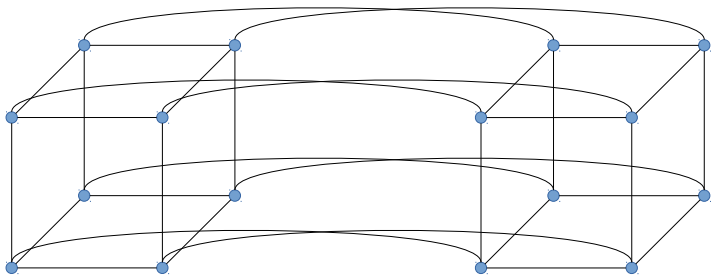


Figure 5.3: Hypercube topology.

The *hypercube* topologies have been widely used for processor interconnections [172, 173]. Figure 5.3 shows an example of the *hypercube* topology with 16 switches where a regular cube has been replicated and the corresponding switches have been directly connected. Moreover, the same replication principle can be applied in order to increase the dimension of the *hypercube* topology. The number of ports per switch needed to connect a switch with the rest of the network is a logarithmic function of the size of the network. However, this topology is a mesh topology which means that it allows any host in the network to communicate with any other host in the network. A downside of the hypercube topology, when compared with the full mesh one, is that usually multiple hops are needed in order to reach the destination, which is not the case for a full-mesh topology because of its direct connections. An advantage of the hypercube topology is its high scalability as only one additional port per switch is needed in order to double the size of the whole network.

In our work, the goal is to create a topology in which we will not use the conventional networking equipment like switches and routers but rather a network where smart NICs will be interconnected and perform the control and the data plane rules of the TRILL protocol.

5.2 Control plane

Each node in the network has a running implementation of the TRILL control plane, while the data plane is implemented on the node's smart NIC. The question that comes up is the choice of the most appropriate topology for our data center network. As we currently have multiple PoP data centers that have 30-80 servers each, we focus our analysis on topologies whose size has this order of magnitude.

The comparison of full-mesh, fat-tree and hypercube topologies is shown in Table 5.1. The derivation of all the equations from Table 5.1 is given in the following subsection. We compare these three topologies in terms of: the total number of links in the topology, the number of TRILL Hello frames sent in the network per minute, the number of TRILL Hello frames sent per link, the number of LSPs sent in the network per minute, the number of LSPs sent over the link and the average distance of the topology.

Table 5.1: Control plane characteristics of different topologies

	Full-mesh	Fat-tree	Hypercube
Total number of links in the topology	$l_{FM}(n) = \frac{n(n-1)}{2}$	$l_{FT}(n) = \frac{1}{2}(\sqrt{\frac{4}{5}}n)^3$	$l_{HC}(n) = \frac{n}{2}log_2 n$
Number of TRILL Hello frames sent in the network per minute	$n_{FM-H}(n) = n(n-1)\lambda_H[\text{min}^{-1}]$	$n_{FT-H}(n) = \lambda_H(\sqrt{\frac{4}{5}}n)^3[\text{min}^{-1}]$	$n_{HC-H}(n) = n\lambda_H \log_2 n[\text{min}^{-1}]$
Number of TRILL Hello frames sent per link	$n_{FM-H_{pl}}(n) = 2\lambda_H[\text{min}^{-1}]$	$n_{FT-H_{pl}}(n) = 2\lambda_H[\text{min}^{-1}]$	$n_{HC-H_{pl}}(n) = 2\lambda_H[\text{min}^{-1}]$
Number of LSPs sent in the network per minute	$n_{FM-LSP}(n) = n(n-1)\lambda_{LSP}[\text{min}^{-1}]$	$n_{FT-LSP}(n) = n(n-1)\lambda_{LSP}[\text{min}^{-1}]$	$n_{HC-LSP}(n) = n(n-1)\lambda_{LSP}[\text{min}^{-1}]$
Number of LSPs generated per link	$n_{FM-LSP_{pl}}(n) = 2\lambda_{LSP}[\text{min}^{-1}]$	$n_{FT-LSP_{pl}}(n) = \frac{5}{4}\lambda_{LSP}\frac{n-1}{\sqrt{\frac{n}{5}}}[\text{min}^{-1}]$	$n_{HC-LSP_{pl}}(n) = 2\lambda_{LSP}\frac{n-1}{log_2 n}[\text{min}^{-1}]$
Average distance of the topology	$\mu_{FM}(n) = 1$	$\mu_{FT}(n) = \frac{\frac{16}{5}n-4\sqrt{\frac{n}{5}}-4}{\frac{4}{5}n-2}$	$\mu_{HC}(n) = \frac{1}{n-1}\sum_{k=1}^{log_2 n}\binom{log_2 n}{k}k$

5.2.1 Calculation of the control plane metrics

Let n designate the number of nodes (where each node has a control plane, while a data plane is implemented on its smart NIC). Let λ_H represent the TRILL Hello frame generation rate (per minute) of a single RBridge on each of its interfaces. Additionally, let λ_{LSP} represent the generation rate of LSPs (per minute) that a single RBridge sends to each RBridge in the network. In the following subsections, we derive the equations from Table 5.1 for the hypercube, the fat-tree and the full-mesh topology.

5.2.1.1 Full-mesh topology

The number of links in the full-mesh topology is equal to the number of nodes n multiplied by the number of links per node $n-1$ and divided by 2 as each of the links is calculated twice in this case:

$$l_{FM}(n) = \frac{n(n-1)}{2}$$

The total number of TRILL Hello frames in the network generated per minute is:

$$n_{FM-H}(n) = n(n-1)\lambda_H[\text{min}^{-1}]$$

The number of TRILL Hello frames sent per link is:

$$n_{FM-H_{pl}}(n) = \frac{n_{FM-H}(n)}{l_{FM}(n)} = 2\lambda_H[\min^{-1}]$$

The total number of LSPs generated in the network is:

$$n_{FM-LSP}(n) = n(n-1)\lambda_{LSP}[\min^{-1}]$$

The number of LSPs generated per link is:

$$n_{FM-LSP_{pl}}(n) = \frac{n_{FM-LSP}(n)}{l_{FM}(n)} = 2\lambda_{LSP}[\min^{-1}]$$

The average distance of the full-mesh topology is equal to 1 as all the RBridges are directly interconnected:

$$\mu_{FM}(n) = 1$$

5.2.1.2 Fat-tree topology

A fat-tree topology that we consider in this thesis is the one defined in [19]. The number of links in the fat-tree topology can be calculated as a sum of the links above and below the aggregate layer. Number of links above the aggregate layer is equal to the number of nodes in the aggregate layer $k\frac{k}{2}$ multiplied by the number of links per node in direction of upper layer $\frac{k}{2}$. The same calculation is done for the edge layer which gives the total number of links in the fat-tree topology to be equal to:

$$l_{FT}(n) = k\frac{k}{2}\frac{k}{2} + k\frac{k}{2}\frac{k}{2} = \frac{k^3}{2} = \frac{1}{2}(\sqrt{\frac{4}{5}n})^3$$

The total number of TRILL Hello frames in the network generated per minute is the sum of such frames generated in the core, the aggregate and the edge layer:

$$\begin{aligned} n_{FT-H}(n) &= [(\frac{k}{2})^2 k + k\frac{k}{2}k + k\frac{k}{2}\frac{k}{2}]\lambda_H[\min^{-1}] \\ &= \lambda_H k^3[\min^{-1}] = \lambda_H(\sqrt{\frac{4}{5}n})^3[\min^{-1}] \end{aligned}$$

The number of TRILL Hello frames sent per link is:

$$n_{FT-H_{pl}}(n) = \frac{n_{FT-H}(n)}{l_{FT}(n)} = 2\lambda_H$$

The total number of LSPs generated in the network is:

$$n_{FT-LSP}(n) = n(n-1)\lambda_{LSP}[\min^{-1}]$$

The number of LSPs generated per link is:

$$n_{FT-LSP_{pl}}(n) = \frac{n_{FT-LSP}(n)}{l_{FT}(n)} = \frac{5}{4}\lambda_{LSP}\frac{n-1}{\sqrt{\frac{n}{5}}}$$

In the fat-tree topology, we assume that the virtual machines are hosted on the same servers as edge RBridges rather than on the servers where the core and aggregate RBridges are. The reason behind this is that, originally, in the fat-tree topology, the hosts are connected on the edge switches and the switches in the aggregation and the core layer were intended only to bring hierarchical interconnection of the edge switches. We follow this assumption in our case where the switches are replaced with the corresponding RBridges in the fat-tree topology. The average distance of the fat-tree topology can be calculated as the sum of distances from one edge RBridge to other edge RBridges in the same pad and the distances from the same RBridge to all the edge RBridges from all the other pads. Afterward, this sum is divided by the number of edge RBridges minus the one from which the distances are calculated:

$$\begin{aligned}\mu_{FT}(n) &= [(\frac{k}{2}-1)2 + (k-1)\frac{k}{2}4]\frac{1}{k\frac{k}{2}-1} \\ &= \frac{4k^2-2k-4}{k^2-2} = \frac{\frac{16}{5}n - 4\sqrt{\frac{n}{5}} - 4}{\frac{4}{5}n-2}\end{aligned}$$

5.2.1.3 Hypercube topology

The number of links in the hypercube topology is equal to the number of nodes n multiplied by the number of links per node $log_2 n$ and divided by 2 as each of the links is calculated twice in this case:

$$l_{HC}(n) = \frac{n}{2}log_2 n$$

The total number of TRILL Hello frames in the network generated per minute is:

$$n_{HC-H}(n) = n\lambda_H \log_2 n[\min^{-1}]$$

The number of TRILL Hello frames sent per minute per link is:

$$n_{HC-H_{pl}}(n) = \frac{n_{HC-H}(n)}{l_{HC}(n)} = 2\lambda_H[\min^{-1}]$$

The total number of LSPs in the network generated per minute is:

$$n_{HC-LSP}(n) = n(n-1)\lambda_{LSP}[\min^{-1}]$$

The number of LSPs generated per link is:

$$n_{HC-LSP_{pl}}(n) = \frac{n_{HC-LSP}(n)}{l_{HC}(n)} = 2\lambda_{LSP}\frac{n-1}{log_2 n}[\min^{-1}]$$

The average distance of the hypercube topology can be calculated in the following way. Firstly, all the nodes of the hypercube are indexed by n bits. The neighboring nodes differ in only one single bit. Consequently, the distance between the two nodes is equal to the number of bits in which their indexes differ. This implicates that the number of neighbors of distance k is equal to $\binom{log_2 n}{k}$. As a result, the average distance from one node to all the other nodes in the network is:

$$\mu_{HC}(n) = \frac{1}{n-1}\sum_{k=1}^{log_2 n}\binom{log_2 n}{k}k$$

This also represents the average distance of the hypercube as hypercube is symmetrical and each of its nodes has the same average distance.

5.2.2 Discussion on parameters used

Each node in the existing implementation [1] is configured to generate a TRILL Hello frame every 3 seconds (20 times per minute) on each of its ports. These frames are used for RBridge neighbor discovery, and this parameter has been configured in Quagga that runs the control plane implementation. Additionally, the LSP generation interval has been set to 30 seconds (2 times per minute). When all these parameters are taken into account, the Figures 5.4, 5.5 and 5.6 show the load and the resiliency information of the data center network. We note here that the vertical bars represent the discrete values for the cases when the topology is "complete" in a way that there are no missing vertices (RBridges) in the topology's structure. For instance, in a trivial case of a hypercube topology where 8 RBridges represent the vertices of the cube, we consider such a topology to be "complete". However, if one

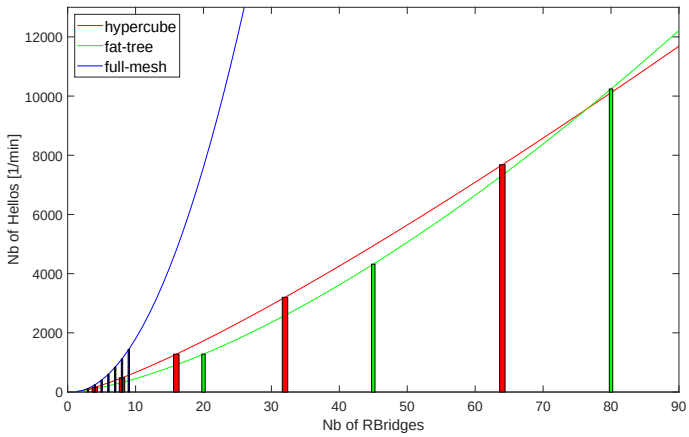


Figure 5.4: Number of TRILL Hello frames per minute in the network.

or more vertices (RBridges) were missing from this topology, we would not consider it to be "complete". Nevertheless, the plotted lines provide a clear visual representation of how the results of "complete" topologies behave when the appropriate topology dimension increases (along with the number of RBridges). As the Kalray MPPA cards that we use have two 40Gbps interfaces, it is possible to split the traffic from each interface with the cables that support 40G QSFP+ to 4X10G SFP+ conversion. In ODP it is possible to process separately the traffic that goes over each of the 4 cables/ports per network interface. This means that we can have 8 cables (from two 40Gbps interfaces) that connect one server (with RBridge implementation and its Kalray card) to the rest of the network. Consequently, this is the same as having 8x10Gbps interfaces on one server. For this reason, in Figures 5.4, 5.5 and 5.6 for the full-mesh network, we have shown only the discrete values for up to 9 RBridges in the network as with Kalray cards we could only create a full-mesh network with one RBridge connected to another 8 RBridges which gives 9 RBridges in total. Similarly, for the other two topologies, we have considered that each RBridge can have up to 8 interfaces depending on the topology's dimension in question. For instance, for the "complete "hypercube topologies, the number total number of RBridges can be 2, 4, 8, 16, 32, 64, 128, etc.

5.2.3 Overhead of the control plane traffic

Figure 5.4 shows how the number of TRILL Hello frames increases much faster with the increase in the number of RBridges in the case of full-mesh network topology when compared with the other two topologies. The reason behind this is that the number of TRILL Hello frames sent per one RBridge (and consequently the total number of TRILL Hello frames in the network) depends directly on the number of first neighbors to each of the RBridges. As in the full-mesh topology each RBridge is connected with all the other RBridges in the network, the consequence of this fact is that it causes a much higher load in the network. Fat-tree topology has a slightly lower number of TRILL Hello frames sent in the network than the hypercube topology when the number of RBridges in the network is lower than 80. From Table 5.1 we can see that the number of TRILL Hello frames sent per link is constant for

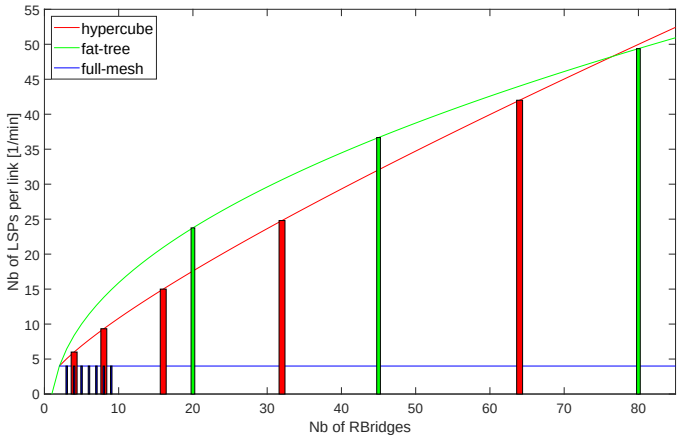


Figure 5.5: The average number of LSPs per minute per link.

all the topologies which is logical because each link represents the direct connection between the two RBridges where both RBridges send the TRILL Hello frames with the λ_H rates.

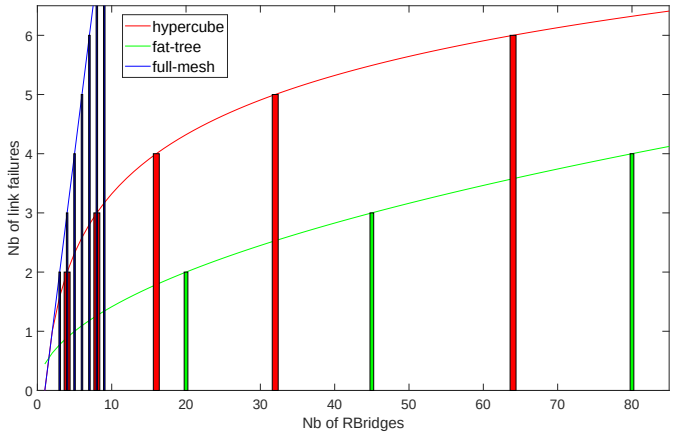


Figure 5.6: Minimum number of link failures to cut-off one RBridge (server).

It can be concluded from Table 5.1 that for all the three topologies, the total number of LSPs generated in the network is the same because each RBridge sends an LSP to all of the RBridges in the TRILL campus. However, the number of LSPs sent per link is different as represented in Figure 5.5. The lowest value is for the case of full-mesh topology because when an LSP is sent, it only needs one hop in order to get to the final destination. This is not the case for non-direct neighbor LSP generations in other topologies. From Figure 5.5 we can also see that for the hypercube topology the number of LSPs is considerably lower in comparison with the fat-tree topology when the number of RBridges in the network is lower than 80.

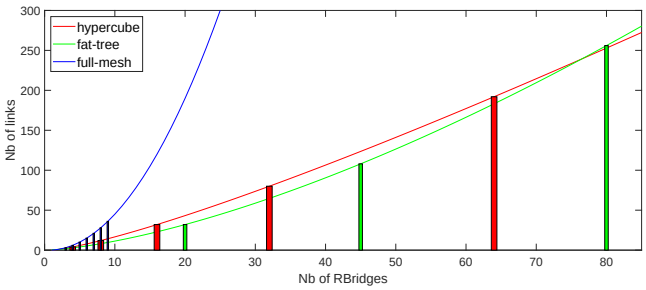


Figure 5.7: Number of links in the topology.

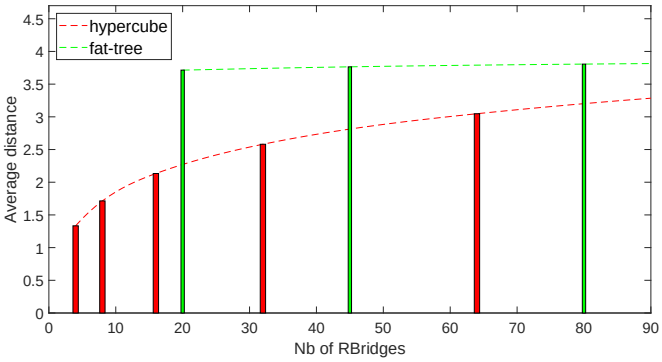


Figure 5.8: Average distance of the topology.

5.2.4 Convergence time

We refer to our previous work [1] where we have shown that in the worst case scenario the new RBridge arrival is detected in $2\frac{1}{\lambda_{LSP}}$ (equal to 60s). Convergence time is independent of the number of nodes in the topology and it is the same for all the topologies. For the DRB (Designated RBridge) the convergence time is around 30s which is equal to the time of the emission of the first LSP by the newly arrived node [1]. The first topology computation occurs after 60s for the newly arrived node and this is the time when this node converges. For all the other nodes, the convergence times are between 45 and 60s due to the asynchronous processing in the DRB.

5.2.5 Resiliency and scalability

Figure 5.6 compares the topologies in terms of resiliency. As expected, the full-mesh topology needs the highest number of link failures to cut-off one RBridge as each one is connected with all the other RBridges in the network. Additionally, the hypercube topology behaves significantly better when compared with the fat-tree topology as it needs around the double of the number of link failures in order to cut-off one switch.

In terms of scalability, the full-mesh topology is the most limited one as it uses up very quickly

the available network interfaces. As previously mentioned, the maximum number of RBridges in the network is *k+1* when using the *k*-port RBridges. The number of links is very similar in hypercube and fat-tree topologies as can be seen in Figure 5.7. However, the average distance of these two topologies differs significantly as can be seen in Figure 5.8. The fat-tree topology has an average distance around 4 for different numbers of RBridges in the TRILL campus, while the hypercube topology has a considerably lower average distance, especially when the number of RBridges is lower than 80. The average distance of full-mesh topology is equal to 1 as all RBridges in the network are directly interconnected and we have omitted to show this trivial case in Figure 5.8. We note here that the average distance for the fat-tree topology has been calculated as the average distance between the edge RBridges as we assume that only the servers that are running the edge RBridges implementations will also be hosting the VMs. Otherwise, it wouldn't make sense to use the fat-tree topology if VMs were hosted behind aggregate or core RBridges as the initial hierarchical structure of this topology, where the traffic is aggregated and load-balanced over the upper layers, would be put in question.

5.2.6 Topology choice

When MPPA smart NICs are used, the full-mesh topology is certainly the best one when we have up to 9 RBridges in the network. However, despite the advantages of the full-mesh topology which has a high resiliency and the smallest average distance, its very limited scalability makes it unsuitable for our use case where we have between 30 and 80 RBridges in the network. In general, a full-mesh topology requires a networking device to have a very high number of network interfaces that grows up linearly with the number of networking devices in the network and this is not the case with the smart NIC cards that we use. On the other hand, a hypercube topology scales very well as it needs only one additional network interface in order to double the size of the whole network. Moreover, a hypercube topology does not have an excessively high control plane load when compared with the other two topologies and it has a significantly better resilience than the fat-tree topology. Additionally, the hypercube topology has a shorter average distance between the nodes when compared with the fat-tree topology. All these facts make a hypercube topology to be the most suitable one for our PoP data center use case with a limited number of servers.

5.3 Conclusion

In this chapter, we provided an analytic model of the impact of different network topologies on the load induced by the control plane in PoP data centers with a limited number of servers. Our solution offloads the data plane packet processing on a smart NIC with parallel processing capability. Packets can be processed at full-duplex line rate (up to 40Gbps) for various packet sizes and reduce latency. The packet processing on the smart NIC allows preserving servers' CPU and memory resources. However, the question that remains is which topology is the best suited for the MPPA smart NICs that we use for our fabric network. We have shown that for our PoP data center use case, the most suitable is the hypercube topology, because it does not have an excessively high control plane load when compared

with the other two topologies, it scales very well and it has a better resilience than the fat-tree while having a significantly shorter average distance between the nodes.

Chapter 6

Conclusion

Summary

With the advent of cloud computing, virtualization technologies and an ever increasing number of applications that rely on them, the amount of data center traffic continues to grow exponentially. Accordingly, the network interface throughputs tend to follow the same trend throughout the years and they currently support the speeds of 40Gbps and higher. Nevertheless, such high interface throughputs do not guarantee faster packet processing in the OS which is limited because of the overheads imposed by the architecture of the network stack. In this thesis, we have investigated and compared many software-based and hardware-based solutions that tend to improve the packet processing performance on server-class network hosts. Many solutions are based on Click Modular router and they try to improve the performance of some of its functions either in software or by using specific hardware. FGPAs and GPUs are mostly used in order to offload part of packet processing and accelerate it. Virtualization of data center networks has brought new requirements such as any-to-any connectivity with full non-blocking fabric, seamless VM mobility, fault tolerance, dynamic scalability, simplified management and private network support. IETF's TRILL protocol is able to fulfill all these needs. It introduces the RBridge devices that are retro-compatible with classical Ethernet bridges by encapsulating the traffic with TRILL header and an additional Outer Ethernet header. Additionally, it allows to convert the Ethernet deployments into a TRILL network by simply replacing classical bridges with

RBridges. However, there is a performance bottleneck when packets are processed on Linux servers with TRILL implementation.

In this thesis, we have also compared the software-based and hardware-based solutions in terms of different properties and we discussed their integration possibilities in virtualized environments, their requirements and their constraints. The goal of this dissertation was to conceive a fabric network solution that increases the packet processing performance on server-class network hosts. For this purpose we used the MPPA hardware acceleration cards. The implementation of the network protocol has been done on the smart NIC by using the ODP API which allows to create portable networking applications. It turns out that such a solution can significantly improve the network performance. Moreover, in order to use the best-suited topology for the interconnection of smart NICs we have considered various criteria and it turns out that hypercube topology is the best for our PoP data center use case.

6.1 Contributions

6.1.1 Fabric network architecture by using hardware acceleration cards

We have started to design our new architecture by identifying the limitations of the existing architecture. As previously mentioned, packet processing has limited performance in Linux because of the generality of the network stack architecture and generality of the device driver design that affect the network throughput. Our goal was to build a fabric network that is highly performant and that offloads packet processing tasks on hardware acceleration cards instead of performing them in the Linux kernel on servers' CPUs. The idea is to replace the classical three-tier network topology with the full-mesh network of links which allows to have predictable network performance and direct connectivity between the nodes. However, we reconsider other network topologies as a part of our third contribution in order to determine which one is best-suited for our PoP data center use case. Such a flat architecture is scalable, resilient and generally decreases the number of hops needed. Additionally, the amount of north-south traffic is reduced because the communication between the VMs hosted on different servers is done over direct links. TRILL protocol is used for communication between the nodes as it provides a better utilization of network links while also providing least-cost pair-wise data forwarding. As each node has a smart NIC, the data plane packet processing is offloaded on this programmable hardware with parallel processing capability. Among the available programming models that were available on the MPPA smart NIC from Kalray, we have chosen to use the ODP API because it allows to reuse the source code of the packet processing application on any other platform that supports the ODP API.

6.1.2 Data plane offloading on a high-speed parallel processing architecture

The communication bottleneck of TRILL data plane implementation in the Linux Bridge Module has been identified. It actually occurs on each TRILL frame hop between different RBridges which requires the Outer Ethernet header that has been added to the original frame to be stripped off . Additionally, the hop count field of the TRILL header gets decreased and the frame gets encapsulated into the new Outer Ethernet header (with the appropriate MAC addresses, after having consulted the

forwarding table). Moreover, there is no fast path solution for the TRILL protocol. For this reason, in order to avoid the communication bottleneck that uses up server's CPU and memory resources, the idea is to offload the data plane packet processing on the MPPA smart NIC. The implementation is done by using the ODP API which permits building protocols from scratch and manipulation of header fields from various networking protocols. The performance evaluation shows that it is possible to perform TRILL frame processing at full-duplex line rate (up to 40Gbps) for various frame sizes while also reducing latency when compared with the existing solution.

6.1.3 Analysis of the fabric network's control plane for the PoP data centers use case

In order to evaluate the impact of different network topologies on the control plane's load, we have performed mathematical analysis that shows how various topology metrics change depending on the number of nodes in the network. We have taken into account that Kalray MPPA cards have two 40Gbps interfaces. Each interface can split its traffic by using cables that support 40G QSFP+ to 4X10G SFP+ conversion. The ODP API allows us to process separately the traffic that goes over each of the 4 cables from one network interface. In other words, 8x10Gbps cables can be connected to one MPPA smart NIC. This fact has motivated us to reconsider a use of full-mesh topology that would, in this case, allow to have a very limited number of cards in the network. Additionally, we considered the use case where each PoP data center has 30-80 servers as is the current state in the production network of Gandi SAS. Thus, despite the high resiliency of the full-mesh topology, its biggest problem is its scalability as it uses up the available network interfaces on the MPPA smart NIC very quickly. A hypercube topology scales very well since it only needs one additional interface in order to double the size of the network. Moreover, it has a better resilience than the fat-tree topology for our use case and it does not have such a high control plane load. It also has a shorter average distance between the nodes which makes it the most suitable topology for our PoP data center use case with a limited number of servers.

6.2 Future works

In this thesis we have presented our fabric network solution that increases the packet processing performance on server-class network hosts by using hardware acceleration cards.

Our short-term future work consists of optimizing the ODP code so that we can process more than 18 million packets per second. This would be a valuable addition especially for applications that use small packet sizes. Such optimizations would assure that a full duplex line-rate (of 40Gbps) can be reached for all packet sizes. For instance, there are many low-level functions from the MPPA API that could do certain memory management functions faster than the ODP functions. Moreover, we plan to obtain multiple acceleration cards and test them at the same time in the public platform of Gandi SAS in order to verify their reliability. Additionally, we will update the code to better suit the production environment needs.

One of the possible mid-term future works would be to use the same TRILL code of ODP API with eventual modifications on some other supported hardware and thus build a hybrid fabric network

that uses different acceleration methods across the network. This can include the ODP-DPDK solution along with ODP implementations on Cavium ThunderX [174], QorIQ [175], KeyStone II System-on-Chip (SoC) [176], Marvell ARMADA SoC [177] or other supported hardware. Platforms that provide higher packet processing performance could be put in the places with higher traffic load. In this way, we could still have some implementations that process packets on server's CPU by the use of ODP-DPDK but these solutions would be used only in places with low traffic loads. However, the problem of efficient packet processing remains a complicated task. Also, the implementation of the protocols requires a certain amount of time even when open standards for packet processing are used such as the ODP API. A transition to P4 programming language is also to be considered especially because of the rising number of hardware accelerators that support it. Also, in general, it is considered to be positioned in a higher level of abstraction when compared with the ODP API.

A possible direction for future research is to build a Virtualized Edge Data Center by using programmable cards. In such circumstances the achievable throughput of programmable hardware accelerators is sufficient to provide services that are hosted close to the end users. The amount of traffic in the edge of the cloud is not excessive when compared to the amount of traffic in large remote data centers. For this reason we consider that the service infrastructure in the edge of the network can use solutions that use programmable hardware which represent an alternative to proprietary cutting-edge network equipment that is better-suited for the core of the large data centers.

Another direction for future research is to consider using hardware acceleration cards for the purpose of DPI (Deep Pakcet Inpsection) because of their proven packet processing performance at high-speeds. In fact, various ML (Machine Learning) algorithms could be implemented on hardware accelerators in order to build an Anti-DDoS (Distributed Denial of Service) solution. The ODP API on packet processors allows to classify the traffic based on the content of various header fields in the received packets. The ML algorithms would learn to recognize new types of attacks which would enable us to mitigate them. A hardware acceleration card would detect the network anomalies and distinguish the attack traffic from regular traffic. In this way, the attack traffic would simply be cut-off from the network while the useful traffic would continue to be transferred across the data center network.

Long-term future work consists of conceiving a sort of hypervisor that is implemented directly on the programmable card. This would allow to spawn VNFs on the hardware accelerator by using a strictly defined part of its processing and memory resources. For instance, an open standard could have an API implemented by using the HLS tools for FPGA cards. For GPUs it would use OpenCL and for other programmable manycore processors it would use their own low-level libraries for the API implementation. Such an API would need to be able to expose the amount of available hardware resources across different platforms in a unified manner so that it is possible to calculate how many VNFs in parallel can be executed on each platform.

Publications

Conferences

[1] D. Cerovic, V. Del Piccolo, A. Amamou and K. Haddadou, "Offloading TRILL on a programmable card," 2016 3rd Smart Cloud Networks & Systems (SCNS), Dubai, 2016, pp. 1-5. doi: 10.1109/SCNS.2016.7870563

[2] V. D. Piccolo, D. Cerovic and K. Haddadou, "Synchronizing BRBs routing information to improve MLTP resiliency," 2017 16th Annual Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net), Budva, 2017, pp. 1-5. doi: 10.1109/MedHocNet.2017.8001646

[3] D. Cerović, V. Del Piccolo, A. Amamou, K. Haddadou and G. Pujolle, "Improving TRILL Mesh Network's Throughput Using Smart NICs," 2018 IEEE/ACM International Symposium on Quality of Service (IWQoS), Banff, Alberta, Canada, 2018

[4] D. Cerović, V. Del Piccolo, A. Amamou, K. Haddadou and G. Pujolle, "Data Plane Offloading on a High-Speed Parallel Processing Architecture," 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), San Francisco, CA, 2018, pp. 229-236. doi: 10.1109/CLOUD.2018.00036

Journals

[5] D. Cerović, V. Del Piccolo, A. Amamou, K. Haddadou and G. Pujolle, "Fast Packet Processing: A Survey," in IEEE Communications Surveys & Tutorials. doi: 10.1109/COMST.2018.2851072

---

[6] D. Cerović, A. Amamou, K. Haddadou and G. Pujolle, "Data center's Parallel Processing Network Architecture Using Smart NICs," under review for IEEE Transactions on Parallel and Distributed Systems

Bibliography

[1] A. Amamou, K. Haddadou, and G. Pujolle, "A TRILL-based multi-tenant data center network," *Computer Networks*, vol. 68, pp. 35 – 53, 2014, communications and Networking in the Cloud. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128614000851

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

[3] 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

[4] L. Rizzo, "Netmap: A Novel Framework for Fast Packet I/O," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 9–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342821.2342830

[5] T. Marian, K. S. Lee, and H. Weatherspoon, "NetSlices: Scalable Multi-core Packet Processing in User-space," in *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '12. New York, NY, USA: ACM, 2012, pp. 27–38. [Online]. Available: http://doi.acm.org/10.1145/2396556.2396563

[6] ntop, "PF_RING," http://www.ntop.org/products/packet-capture/pf_ring/, (Accessed on 01/11/2018).

[7] Intel, "Impressive Packet Processing Performance Enables Greater Workload Consolidation," https://www.techonline.com/electrical-engineers/education-training/tech-papers/4415080/Impressive-Packet-Processing-Performance-Enables-Greater-Workload-Consolidation, (Accessed on 01/11/2018).

[8] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-accelerated Software Router," in *Proceedings of the ACM SIGCOMM 2010 Conference*, ser. SIGCOMM '10. New York, NY, USA: ACM, 2010, pp. 195–206. [Online]. Available: http://doi.acm.org/10.1145/1851182.1851207

[9] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang, and K. Park, "APUNet: Revitalizing GPU as Packet Processing Accelerator," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 83–96. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/go

[10] P. Bellows, J. Flidr, T. Lehman, B. Schott, and K. D. Underwood, "GRIP: a reconfigurable architecture for host-based gigabit-rate packet processing," in *Field-Programmable Custom Computing Machines, 2002. Proceedings. 10th Annual IEEE Symposium on*, 2002, pp. 121–130.

[11] M. B. Anwer, M. Motiwala, M. b. Tariq, and N. Feamster, "SwitchBlade: A Platform for Rapid Deployment of Network Protocols on Programmable Hardware," in *Proceedings of the ACM SIGCOMM 2010 Conference*, ser. SIGCOMM '10. New York, NY, USA: ACM, 2010, pp. 183–194. [Online]. Available: http://doi.acm.org/10.1145/1851182.1851206

[12] Open vSwitch* Enables SDN and NFV Transformation. Intel. [Online]. Available: https://networkbuilders.intel.com/docs/open-vswitch-enables-sdn-and-nfv-transformation-paper.pdf

[13] B. Li, K. Tan, L. Luo, R. Luo, Y. Peng, N. Xu, Y. Xiong, P. Chen, Y. Xiong, and P. Cheng, "ClickNP: Highly Flexible and High-performance Network Processing with Reconfigurable Hardware." ACM, July 2016. [Online]. Available: https://www.microsoft.com/en-us/research/publication/clicknp-highly-flexible-high-performance-network-processing-reconfigurable-hardware/

[14] "OpenFastPath - Technical Overview," http://www.openfastpath.org/index.php/service/technicaloverview/, (Accessed on 01/11/2018).

[15] "Accelerate networking innovation through programmable data plane - Removing switches from datacenters with TRILL/VNT and smartNIC," http://pres.gandi.net/ocpeu14/ocpeu14.pdf, (Accessed on 01/11/2018).

[16] B. D. de Dinechin, "Kalray MPPA : Massively parallel processor array: Revisiting DSP acceleration with the Kalray MPPA Manycore processor," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, Aug 2015, pp. 1–27.

[17] E. Warnicke, "OpenDataPlane (ODP) Users-Guide," https://www.opendataplane.org/application-writers/users-guide/, (Accessed on 01/11/2018).

[18] D. Cerović, V. D. Piccolo, A. Amamou, K. Haddadou, and G. Pujolle, "Data plane offloading on a high-speed parallel processing architecture," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, July 2018, pp. 229–236.

[19] M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 63–74, Aug. 2008. [Online]. Available: http://doi.acm.org/10.1145/1402946.1402967

[20] C. V. Networking, "Cisco Global Cloud Index: Forecast and Methodology 2016-2021 (White Paper)," 2018.

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

[22] "Intel® Ethernet Switch FM10000," https://goo.gl/eCdzqu, (Accessed on 01/11/2018).

[23] "Broadcom's Tomahawk® 3 Ethernet switch chip delivers 12.8 Tb/s of speed in a single 16 nm device," https://www.broadcom.com/blog/broadcom-s-tomahawk-3-ethernet-switch-chip-delivers-12-8-tbps-of-speed-in-a-single-16-nm-device, (Accessed on 01/11/2018).

[24] The Future Is 40 Gigabit Ethernet. Cisco. [Online]. Available: http://www.cisco.com/c/dam/en/us/products/collateral/switches/catalyst-6500-series-switches/white-paper-c11-737238.pdf

[25] I. Marinos, R. N. Watson, and M. Handley, "Network stack specialization for performance," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 175–186. [Online]. Available: http://doi.acm.org/10.1145/2619239.2626311

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

[27] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 5–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=2772722.2772727

[28] Intel, "DPDK: Data plane development kit." http://dpdk.org/, (Accessed on 01/11/2018).

[29] Solarflare, "OpenOnload," http://www.openonload.org/, (Accessed on 01/11/2018).

[30] "PACKET_MMAP. Linux Kernel Contributors. ," https://goo.gl/2YqJyK, (Accessed on 01/11/2018).

[31] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "Routebricks: Exploiting parallelism to scale software routers," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 15–28. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629578

[32] W. Sun and R. Ricci, "Fast and Flexible: Parallel Packet Processing with GPUs and Click," in *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 25–36. [Online]. Available: http://dl.acm.org/citation.cfm?id=2537857.2537861

[33] J. Kim, S. Huh, K. Jang, K. Park, and S. B. Moon, "The power of batching in the Click modular router," in *Asia-Pacific Workshop on Systems, APSys '12, Seoul, Republic of Korea, July 23-24, 2012*. ACM, 2012, p. 14. [Online]. Available: http://doi.acm.org/10.1145/2349896.2349910

[34] B. Chen and R. Morris, "Flexible Control of Parallelism in a Multiprocessor PC Router." in *USENIX Annual Technical Conference, General Track*, 2001, pp. 333–346.

[35] 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, Sept 2004.

[36] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "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

[37] J. C. Mogul, P. Yalag, J. Tourrilhes, R. Mcgeer, S. Banerjee, T. Connors, and P. Sharma, "API design challenges for open router platforms on proprietary hardware," in *in Proc. HotNets-VII*, 2008.

[38] O. E. Ferkouss, I. Snaiki, O. Mounaouar, H. Dahmouni, R. B. Ali, Y. Lemieux, and C. Omar, "A 100Gig Network Processor platform for Openflow," in *2011 7th International Conference on Network and Service Management*, Oct 2011, pp. 1–4.

[39] Y. K. Chang and F.-C. Kuo, "Towards optimized packet processing for multithreaded network processor," in *2010 International Conference on High Performance Switching and Routing*, June 2010, pp. 127–132.

[40] T. Wolf and M. A. Franklin, "Performance models for network processor design," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 6, pp. 548–561, June 2006.

[41] E. Rubow, R. McGeer, J. C. Mogul, and A. Vahdat, "Chimpp: a click-based programming and simulation environment for reconfigurable networking hardware," in *Proceedings of the 2010 ACM/IEEE Symposium on Architecture for Networking and Communications Systems, ANCS 2010, San Diego, California, USA, October 25-26, 2010*, B. Lin, J. C. Mogul, and R. R. Iyer, Eds. ACM, 2010, p. 36. [Online]. Available: http://doi.acm.org/10.1145/1872007.1872052

[42] J. Naous, G. Gibb, S. Bolouki, and N. McKeown, "NetFPGA: Reusable Router Architecture for Experimental Research," in *Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow*, ser. PRESTO '08. New York, NY, USA: ACM, 2008, pp. 1–7. [Online]. Available: http://doi.acm.org/10.1145/1397718.1397720

[43] M. Attig and G. Brebner, "400 Gb/s Programmable Packet Parsing on a Single FPGA," in *Architectures for Networking and Communications Systems (ANCS), 2011 Seventh ACM/IEEE Symposium on*, Oct 2011, pp. 12–23.

[44] A. Bitar, M. S. Abdelfattah, and V. Betz, "Bringing programmability to the data plane: Packet processing with a NoC-enhanced FPGA," in *Field Programmable Technology (FPT), 2015 International Conference on*, Dec 2015, pp. 24–31.

[45] 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. [Online]. Available: https://doi.org/10.1109/TNET.2016.2597163

[46] Berten, "White paper - GPU vs FPGA Performance Comparison," https://goo.gl/raKHdS, (Accessed on 01/11/2018).

[47] L. Rizzo and G. Lettieri, "VALE, a Switched Ethernet for Virtual Machines," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '12. New York, NY, USA: ACM, 2012, pp. 61–72. [Online]. Available: http://doi.acm.org/10.1145/2413176.2413185

[48] L. Rizzo, G. Lettieri, and V. Maffione, "Speeding up packet I/O in virtual machines," in *Architectures for Networking and Communications Systems*, Oct 2013, pp. 47–58.

[49] S. Garzarella, G. Lettieri, and L. Rizzo, "Virtual device passthrough for high speed VM networking," in *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, May 2015, pp. 99–110.

[50] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms," *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 34–47, March 2015.

[51] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker, "Extending networking into the virtualization layer," in *Proc. of workshop on Hot Topics in Networks (HotNets-VIII)*, 2009.

[52] G. Robin, "Open vSwitch with DPDK Overview. Intel." https://software.intel.com/en-us/articles/open-vswitch-with-dpdk-overview, (Accessed on 01/11/2018).

[53] W. Xia, P. Zhao, Y. Wen, and H. Xie, "A Survey on Data Center Networking (DCN): Infrastructure and Operations," *IEEE Communications Surveys Tutorials*, vol. 19, no. 1, pp. 640–656, Firstquarter 2017.

[54] D. E. E. 3rd, D. G. Dutt, S. Gai, R. Perlman, and A. Ghanwani, "Routing Bridges (RBridges): Base Protocol Specification," RFC 6325, Jul. 2011. [Online]. Available: https://rfc-editor.org/rfc/rfc6325.txt

[55] "ktrill - TRILL implementation in the Linux Kernel," accessed 7 September 2017. [Online]. Available: https://github.com/Gandi/ktrill/tree/devel

[56] D. Cerović, V. D. Piccolo, A. Amamou, K. Haddadou, and G. Pujolle, "Fast packet processing: A survey," *IEEE Communications Surveys Tutorials*, 2018.

[57] "The Juniper Networks QFabric Architecture: A Revolution in Data Center Network Design Flattening the Data Center Architecture," https://www.slideshare.net/TechnologyBIZ/juniper-networks-q-fabric-architecture, (Accessed on 01/11/2018).

[58] S. Hooda, S. Kapadia, and P. Krishnan, *Using TRILL, FabricPath, and VXLAN: Designing Massively Scalable Data Centers (MSDC) with Overlays*, ser. Networking Technology. Pearson Education, 2014. [Online]. Available: https://books.google.fr/books?id=k5nFAgAAQBAJ

[59] "Cisco Data Center Spine-and-Leaf Architecture: Design Overview White Paper," https://www.cisco.com/c/en/us/products/collateral/switches/nexus-7000-series-switches/white-paper-c11-737022.html, (Accessed on 26/10/2018).

[60] "Ethernet Fabrics and Brocade VCS Technology: Networking for the Virtualized Data Center," https://cstenet.com/brocade/files/Brocade%20Data%20Center.pdf, (Accessed on 01/11/2018).

[61] "IEEE 802.1aq - Shortest Path Bridging," http://www.ieee802.org/1/pages/802.1aq.html, (Accessed on 01/11/2018).

[62] D. Allan and N. Bragg, *802.1aq Shortest Path Bridging Design and Evolution: The Architect's Perspective*. Wiley, 2012. [Online]. Available: https://books.google.fr/books?id=NRuOJ7xdcHMC

[63] R. van der Pol, "TRILL and IEEE 802.1aq Overview," https://kirk.rvdp.org/publications/TRILL-SPB.pdf, (Accessed on 01/11/2018).

[64] C. R. Meiners, A. X. Liu, and E. Torng, *Hardware Based Packet Classification for High Speed Internet Routers*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[65] R. Ramaswamy, N. Weng, and T. Wolf, "Characterizing network processing delay," in *Global Telecommunications Conference, 2004. GLOBECOM '04. IEEE*, vol. 3, Nov 2004, pp. 1629–1634 Vol.3.

[66] H. Asai, "Where Are the Bottlenecks in Software Packet Processing and Forwarding?: Towards High-performance Network Operating Systems," in *Proceedings of The Ninth International Conference on Future Internet Technologies*, ser. CFI '14. New York, NY, USA: ACM, 2014, pp. 5:1–5:6. [Online]. Available: http://doi.acm.org/10.1145/2619287.2619300

[67] L. Rizzo, L. Deri, and A. Cardigliano, "10 Gbit/s Line Rate Packet Processing Using Commodity Hardware: Survey and new Proposals," 2012.

[68] S. Networking, "Eliminating the Receive Processing Bottleneck—Introducing RSS," *Microsoft WinHEC (April 2004)*, 2004.

[69] L. Deri, "nCap: wire-speed packet capture and transmission," in *Workshop on End-to-End Monitoring Techniques and Services, 2005.*, May 2005, pp. 47–55.

[70] First the Tick, Now the Tock Next Generation Intel Microarchitecture (Nehalem). Intel. [Online]. Available: https://goo.gl/KK6DH5

[71] N. Egi, M. Dobrescu, J. Du, K. Argyraki, B. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, L. Mathy *et al.*, "Understanding the packet processing capability of multi-core servers," Intel Technical Report, Tech. Rep., 2009.

[72] M. Miao, W. Cheng, F. Ren, and J. Xie, "Smart Batching: A Load-Sensitive Self-Tuning Packet I/O Using Dynamic Batch Sizing," in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, Dec 2016, pp. 726–733.

[73] L. Rizzo, "OVS: accelerating the datapath through netmap/VALE," http://openvswitch.org/support/ovscon2014/18/1630-ovs-rizzo-talk.pdf, (Accessed on 01/11/2018).

[74] L. Deri, N. S. P. A, V. D. B. Km, and L. L. Figuretta, "Improving Passive Packet Capture: Beyond Device Polling," in *In Proceedings of SANE 2004*.

[75] 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

[76] J. H. Salim, "When NAPI comes to town," in *Linux 2005 Conf*, 2005.

[77] 6WIND, "6WINDGate Packet Processing Software For COTS Servers Scalable Data Plane Performance For Multicore Platforms," http://www.6wind.com/wp-content/uploads/2016/08/6WINDGate-Data-Sheet.pdf, (Accessed on 01/11/2018).

[78] J. Nickolls, "GPU parallel computing architecture and CUDA programming model," in *2007 IEEE Hot Chips 19 Symposium (HCS)*, Aug 2007, pp. 1–12.

[79] 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

[80] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen, "Raising the Bar for Using GPUs in Software Packet Processing," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 409–423. [Online]. Available: http://dl.acm.org/citation.cfm?id=2789770.2789799

[81] V. Volkov, "Understanding latency hiding on gpus," Ph.D. dissertation, EECS Department, University of California, Berkeley, Aug 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-143.html

[82] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis, "GASPP: A GPU-Accelerated Stateful Packet Processing Framework," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, 2014, pp. 321–332. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/vasiliadis

[83] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, "Comparison of frameworks for high-performance packet IO," in *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, May 2015, pp. 29–38.

[84] T. Hudek and D. MacMichael, "Introduction to the NDIS PacketDirect Provider Interface," https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-ndis-pdpi, (Accessed on 01/11/2018).

[85] S. Gueron, "Speeding up CRC32C computations with Intel CRC32 instruction," *Information Processing Letters*, vol. 112, no. 5, pp. 179 – 185, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S002001901100319X

[86] Intel, "Intel®SSE4 Programming Reference, p.61," https://software.intel.com/sites/default/files/m/8/b/8/D9156103.pdf, (Accessed on 01/11/2018).

[87] "Click Modular Router," https://github.com/kohler/click/, (Accessed on 01/11/2018).

[88] "RouteBricks," http://routebricks.org/code.html, (Accessed on 01/11/2018).

[89] "FastClick," https://gitlab.run.montefiore.ulg.ac.be/sdn-pp/fastclick/, (Accessed on 01/11/2018).

[90] "Snap," https://github.com/wbsun/snap, (Accessed on 01/11/2018).

[91] "netmap," https://github.com/luigirizzo/netmap, (Accessed on 01/11/2018).

[92] "NetSlice," http://fireless.cs.cornell.edu/netslice/, (Accessed on 01/11/2018).

[93] "PF_RING," https://github.com/ntop/PF_RING, (Accessed on 01/11/2018).

[94] "DPDK," http://dpdk.org/download, (Accessed on 01/11/2018).

[95] "PacketShader," https://github.com/ANLAB-KAIST/Packet-IO-Engine, (Accessed on 01/11/2018).

[96] "PacketShader website," https://shader.kaist.edu/packetshader/, (Accessed on 01/11/2018).

[97] M. Bourguiba, K. Haddadou, and G. Pujolle, "A Container-Based Fast Bridge for Virtual Routers on Commodity Hardware," in *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*, Dec 2010, pp. 1–6.

[98] M. Bourguiba, K. Haddadou, I. E. Korbi, and G. Pujolle, "A Container-Based I/O for Virtual Routers: Experimental and Analytical Evaluations," in *2011 IEEE International Conference on Communications (ICC)*, June 2011, pp. 1–6.

[99] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 164–177. [Online]. Available: http://doi.acm.org/10.1145/945445.945462

[100] M. Bourguiba, K. Haddadou, I. E. Korbi, and G. Pujolle, "Improving Network I/O Virtualization for Cloud Computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 673–681, March 2014.

[101] F. Bellard, "QEMU, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[102] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: the Linux Virtual Machine Monitor," in *In Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07*, 2007.

[103] V. Maffione, L. Rizzo, and G. Lettieri, "Flexible virtual machine networking using netmap passthrough," in *2016 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, June 2016, pp. 1–6.

[104] 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

[105] J. Tseng, R. Wang, J. Tsai, Y. Wang, and T.-Y. C. Tai, "Accelerating Open vSwitch with Integrated GPU," in *Proceedings of the Workshop on Kernel-Bypass Networks*, ser. KBNets '17. New York, NY, USA: ACM, 2017, pp. 7–12. [Online]. Available: http://doi.acm.org/10.1145/3098583.3098585

[106] J. Tseng, R. Wang, J. Tsai, S. Edupuganti, A. W. Min, S. Woo, S. Junkins, and T. Y. C. Tai, "Exploiting integrated GPUs for network packet processing workloads," in *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, June 2016, pp. 161–165.

[107] 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 *2016 Fifth European Workshop on Software-Defined Networks (EWSDN)*, Oct 2016, pp. 19–24.

[108] E. Kohler, "The Click modular router [Ph. D. Thesis]. Department of Electrical Engineering and Computer Science," 2001.

[109] "Altera SDK for OpenCL," http://www.altera.com/, (Accessed on 01/11/2018).

[110] "SDAccel Development Environment," http://www.xilinx.com/, (Accessed on 01/11/2018).

[111] "Vivado Design Suite," http://www.xilinx.com/, (Accessed on 01/11/2018).

[112] "NVIDIA CUDA SDK 3.0." https://developer.nvidia.com/cuda-toolkit-30-downloads, (Accessed on 01/11/2018).

[113] "Intel - Linux ixgbe* Base Driver Overview and Installation." https://www.intel.com/content/www/us/en/support/network-and-i-o/ethernet-products/000005688.html, (Accessed on 01/11/2018).

[114] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 13–24. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665678

[115] "Syskonnect gigabit ethernet card," http://www.syskonnect.com/, (Accessed on 01/11/2018).

[116] "XMACII chipset," https://goo.gl/zUrcnr, (Accessed on 01/11/2018).

[117] 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," *The Linux Foundation*, pp. 1–19, 2016.

[118] "fd.io," https://fd.io/, (Accessed on 01/11/2018).

[119] E. Warnicke, "fd.io Intro," https://wiki.fd.io/view/File:Fdio_intro_2016-03-10.pptx, (Accessed on 01/11/2018).

[120] "Vector Packet Processing (VPP)," https://wiki.fd.io/view/VPP, (Accessed on 01/11/2018).

[121] "VPP/What is VPP?" https://goo.gl/q1Jqhw, (Accessed on 01/11/2018).

[122] "VPP on OpenDataPlane enabled SmartNICs," goo.gl/EV1Qep, (Accessed on 01/11/2018).

[123] "OpenDataPlane (ODP) Project," https://www.opendataplane.org/, (Accessed on 01/11/2018).

[124] "PCIe NIC optimised implementation (odp-dpdk)," https://git.linaro.org/lng/odp-dpdk.git, (Accessed on 01/11/2018).

[125] "OpenDataPlane DPDK platform implementation," https://github.com/Linaro/odp-dpdk, (Accessed on 01/11/2018).

[126] "OpenFastPath," http://www.openfastpath.org/, (Accessed on 01/11/2018).

[127] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "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

[128] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon, "P4FPGA: A Rapid Prototyping Framework for P4," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17. New York, NY, USA: ACM, 2017, pp. 122–135. [Online]. Available: http://doi.acm.org/10.1145/3050220.3050234

[129] P. Li and Y. Luo, "P4GPU: Accelerate Packet Processing of a P4 Program with a CPU-GPU Heterogeneous Architecture," in *Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '16. New York, NY, USA: ACM, 2016, pp. 125–126. [Online]. Available: http://doi.acm.org/10.1145/2881025.2889480

[130] P. Benácek, V. Pu, and H. Kubátová, "P4-to-VHDL: Automatic Generation of 100 Gbps Packet Parsers," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2016, pp. 148–155.

[131] A. Sivaraman, C. Kim, R. Krishnamoorthy, A. Dixit, and M. Budiu, "DC.P4: Programming the Forwarding Plane of a Data-center Switch," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15. New York, NY, USA: ACM, 2015, pp. 2:1–2:8. [Online]. Available: http://doi.acm.org/10.1145/2774993.2775007

[132] S. Laki, D. Horpácsi, P. Vörös, R. Kitlei, D. Leskó, and M. Tejfel, "High Speed Packet Forwarding Compiled from Protocol Independent Data Plane Specifications," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 629–630. [Online]. Available: http://doi.acm.org/10.1145/2934872.2959080

[133] A. Bhardwaj, A. Shree, V. B. Reddy, and S. Bansal, "A Preliminary Performance Model for Optimizing Software Packet Processing Pipelines," in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, ser. APSys '17. New York, NY, USA: ACM, 2017, pp. 26:1–26:7. [Online]. Available: http://doi.acm.org/10.1145/3124680.3124747

[134] 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

[135] S. Pontarelli, M. Bonola, G. Bianchi, A. Capone, and C. Cascone, "Stateful OpenFlow: Hardware proof of concept," in *2015 IEEE 16th International Conference on High Performance Switching and Routing (HPSR)*, July 2015, pp. 1–8.

[136] "BESS (Berkeley Extensible Software Switch)," http://span.cs.berkeley.edu/bess.html, (Accessed on 01/11/2018).

[137] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "SoftNIC: A Software NIC to Augment Hardware," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155, May 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html

[138] "QFabric System Overview," https://www.juniper.net/documentation/en_US/junos/topics/concept/qfabric-overview.html, (Accessed on 26/10/2018).

[139] "Cisco FabricPath," https://www.cisco.com/c/en/us/solutions/data-center-virtualization/fabricpath/index.html, (Accessed on 26/10/2018).

[140] "VCS Fabrics overview," http://wwwaem.brocade.com/content/html/en/administration-guide/nos-700-adminguide/GUID-62474CEC-B04E-4B52-B527-27B17D8A4044.html, (Accessed on 26/10/2018).

[141] "Shortest Path Bridging IEEE 802.1aq," http://sites.ieee.org/houston/files/2015/12/Team-Brian-Miller-Yuri-Spillman4.pdf, (Accessed on 01/11/2018).

[142] "The Great Debate: TRILL Versus 802.1aq (SBP)," https://www.nanog.org/meetings/nanog50/presentations/Monday/NANOG50.Talk63.NANOG50_TRILL-SPB-Debate-Roisman.pdf, (Accessed on 01/11/2018).

[143] Y. Liao, D. Yin, and L. Gao, "PdP: Parallelizing Data Plane in Virtual Network Substrate," in *Proceedings of the 1st ACM Workshop on Virtualized Infrastructure Systems and Architectures*, ser. VISA '09. New York, NY, USA: ACM, 2009, pp. 9–18. [Online]. Available: http://doi.acm.org/10.1145/1592648.1592651

[144] J. Kim, S. Huh, K. Jang, K. Park, and S. Moon, "The Power of Batching in the Click Modular Router," in *Proceedings of the Asia-Pacific Workshop on Systems*, ser. APSYS '12. New York, NY, USA: ACM, 2012, pp. 14:1–14:6. [Online]. Available: http://doi.acm.org/10.1145/2349896.2349910

[145] L. Rizzo, M. Carbone, and G. Catalli, "Transparent acceleration of software packet forwarding using netmap," in *2012 Proceedings IEEE INFOCOM*, March 2012, pp. 2471–2479.

[146] D. Blythe, "Rise of the Graphics Processor," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 761–778, May 2008.

[147] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware,"

Computer Graphics Forum, vol. 26, no. 1, pp. 80–113, 2007. [Online]. Available: http://dx.doi.org/10.1111/j.1467-8659.2007.01012.x

[148] E. Papadogiannaki, L. Koromilas, G. Vasiliadis, and S. Ioannidis, "Efficient Software Packet Processing on Heterogeneous and Asymmetric Hardware Architectures," *IEEE/ACM Transactions on Networking*, vol. 25, no. 3, pp. 1593–1606, June 2017.

[149] Z. Zheng, J. Bi, H. Yu, C. Sun, and J. Wu, "BLOP: Batch-Level Order Preserving for GPU-Accelerated Packet Processing," in *Proceedings of the SIGCOMM Posters and Demos*, ser. SIGCOMM Posters and Demos '17. New York, NY, USA: ACM, 2017, pp. 136–137. [Online]. Available: http://doi.acm.org/10.1145/3123878.3132013

[150] F. Fusco, M. Vlachos, X. Dimitropoulos, and L. Deri, "Indexing million of packets per second using gpus," in *Proceedings of the 2013 Conference on Internet Measurement Conference*, ser. IMC '13. New York, NY, USA: ACM, 2013, pp. 327–332. [Online]. Available: http://doi.acm.org/10.1145/2504730.2504756

[151] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[152] M. Garland, "Parallel computing with CUDA," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, April 2010, pp. 1–1.

[153] Y. Zhu, Y. Deng, and Y. Chen, "Hermes: An integrated CPU/GPU microarchitecture for IP routing," in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2011, pp. 1044–1049.

[154] S. Mu, X. Zhang, N. Zhang, J. Lu, Y. S. Deng, and S. Zhang, "IP routing processing with graphic processors," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, March 2010, pp. 93–98.

[155] B. Betkaoui, D. B. Thomas, and W. Luk, "Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing," in *Field-Programmable Technology (FPT), 2010 International Conference on*, Dec 2010, pp. 94–101.

[156] S. Mittal and J. S. Vetter, "A Survey of Methods for Analyzing and Improving GPU Energy Efficiency," *ACM Comput. Surv.*, vol. 47, no. 2, pp. 19:1–19:23, Aug. 2014. [Online]. Available: http://doi.acm.org/10.1145/2636342

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

[158] T. Rinta-Aho, M. Karlstedt, and M. P. Desai, "The Click2NetFPGA Toolchain," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 7–7. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342821.2342828

[159] 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 Transactions on Education*, vol. 51, no. 3, pp. 364–369, Aug 2008.

[160] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo, "NetFPGA–An Open Platform for Gigabit-Rate Network Switching and Routing," in *Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, ser. MSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 160–161. [Online]. Available: http://dx.doi.org/10.1109/MSE.2007.69

[161] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "NetFPGA SUME: Toward Research Commodity 100Gb/s," 2014.

[162] N. Zilberman, Y. Audzevich, G. Kalogeridou, N. M. Bojan, J. Zhang, and A. W. Moore, "NetFPGA - rapid prototyping of high bandwidth devices in open source," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2015, pp. 1–1.

[163] M. Labrecque, J. G. Steffan, G. Salmon, M. Ghobadi, and Y. Ganjali, "NetThreads: Programming NetFPGA with threaded software," 2009.

[164] P. Varga, L. Kovács, T. Tóthfalusi, and P. Orosz, "C-GEP: 100 Gbit/s capable, FPGA-based, reconfigurable networking equipment," in *2015 IEEE 16th International Conference on High Performance Switching and Routing (HPSR)*, July 2015, pp. 1–6.

[165] B. Wheeler, "A new era of network processing," *The Linley Group, Tech. Rep*, 2013.

[166] J. Shim, J. Kim, K. Lee, and S. Moon, "Knapp: A packet processing framework for manycore accelerators," in *2017 IEEE 3rd International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB)*, Feb 2017, pp. 57–64.

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

[168] "OpenDataPlane port for the MPPA platform," accessed 16 September 2017. [Online]. Available: https://github.com/kalray/odp-mppa

[169] A. Amamou, M. Bourguiba, K. Haddadou, and G. Pujolle, "DBA-VM: Dynamic bandwidth allocator for virtual machines," in *2012 IEEE Symposium on Computers and Communications (ISCC)*, July 2012, pp. 000 713–000 718.

[170] Intel® Ethernet Controller 10 Gigabit and 40 Gigabit XL710 Brief . [Online]. Available: https://www.intel.com/content/www/us/en/embedded/products/networking/xl710-10-40-gbe-controller-brief.html?asset=8353

[171] C. E. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, vol. C-34, no. 10, pp. 892–901, Oct 1985.

[172] Y. Liu, J. Han, and H. Du, "A hypercube-based scalable interconnection network for massively parallel computing," *Journal of Computers*, vol. 3, no. 10, pp. 58–65, 2008.

[173] A. Louri, B. Weech, and C. Neocleous, "A spanning multichannel linked hypercube: a gradually scalable optical interconnection network for massively parallel computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 5, pp. 497–512, May 1998.

[174] "Cavium ThunderX implimentation of ODP," https://github.com/Linaro/odp-thunderx, (Accessed on 01/11/2018).

[175] "QorIQ - sdk/odp.git - Open Data Plane Interface Implementation," http://git.freescale.com/git/cgit.cgi/ppc/sdk/odp.git/?h=fsl_odp_v16.07_qoriq, (Accessed on 01/11/2018).

[176] "TIs Keystone2 with multi core System-on-Chip Cortex A15 and TMS320C66x DSP hardware acceleration for ODP," https://git.linaro.org/lng/odp-keystone2.git, (Accessed on 01/11/2018).

[177] "OpenDataPlane Marvell implementation - odp-marvell," https://github.com/MarvellEmbeddedProcessors/odp-marvell, (Accessed on 01/11/2018).

[178] D. E. E. 3rd, R. Perlman, A. Ghanwani, H. Yang, and V. Manral, "Transparent Interconnection of Lots of Links (TRILL): Adjacency," RFC 7177, May 2014. [Online]. Available: https://rfc-editor.org/rfc/rfc7177.txt

[179] D. E. E. 3rd, M. Zhang, R. Perlman, A. Banerjee, A. Ghanwani, and S. Gupta, "Transparent Interconnection of Lots of Links (TRILL): Clarifications, Corrections, and Updates," RFC 7780, Feb. 2016. [Online]. Available: https://rfc-editor.org/rfc/rfc7780.txt

[180] ISO/IEC 10589:2002(E), "Information technology — Telecommunications and information exchange between systems — Intermediate System to Intermediate System intra-domain routeing information exchange protocol for use in conjunction with the protocol for providing the connectionless-mode network service (ISO 8473)," International Organization for Standardization, Geneva, CH, Standard, Nov. 2002.

Appendices

Appendix A

TRILL protocol

A.1 Introduction

TRILL protocol [54, 178, 179] is an IETF protocol which aims to overcome the limitations of conventional Ethernet networks while introducing the advantages of network layer protocols. It provides least-cost pair-wise data forwarding without configuration, safe forwarding during the periods of temporary loops and it supports the multipathing of both unicast and multicast traffic [54]. Additionally, TRILL allows to create a cloud of links that acts as a single IP subnet from the IP nodes point of view. Nodes within this cloud can migrate without changing their IP addresses. Despite the simplicity of the spanning tree algorithm in the conventional Ethernet networks, there are certain limitations. In fact, the bandwidth across the subnet is limited as the traffic is concentrated on the spanning tree path (even in the case when more direct paths are available). For this reason, TRILL protocol does not use spanning tree but instead uses the shortest path between any two points with multipath capability in the case when several paths are the shortest ones. In fact, TRILL uses an extension of IS-IS [180] as its routing protocol because it has several advantages [54]. Firstly, it runs directly on layer 2 which allows it to run without configuration (as there is no need to assign the IP addresses). Another advantage is that IS-IS protocol can be easily extended through definition of new TLV (type-length-value) data elements and sub-elements that contain TRILL information [54]. A conversion of a traditional Ethernet deployment into a TRILL network is simple. It makes the cloud more stable with more effective usage of the available bandwidth and it is done by replacing a set of conventional bridges with the RBridges.

RBridges are devices which implement TRILL protocol and their role is similar to that of the conventional customer bridges. RBridges are addressed with 2B nicknames, which act as the abbreviations for the IS-IS IDs of Rbridges to achieve more compact encoding [54]. Each RBridge has a

unique nickname within the TRILL network (called a campus). Each RBridge in the TRILL campus can choose its own nickname or the nicknames may be configured by an administrator. In the case where a nickname of an RBridge is not unique, then the RBridge with the numerically higher IS-IS ID (LAN ID) keeps the nickname or if their priorities are equal, then the RBridge with the numerically higher IS-IS System ID keeps the nickname while the other RBridge needs to select a new nickname [54]. Unicast forwarding tables of RBridges are significantly smaller than the tables in conventional customer bridges as they consist of all the entries which describe RBridges in a TRILL campus, rather than all the entries which describe end nodes.

Whenever an Ethernet frame encounters the first RBridge (called the ingress RBridge) on its way to a destination, it gets encapsulated. The encapsulation consists of adding a TRILL header and an Outer Ethernet header around the original packet with its (Inner) Ethernet header. The ingress RBridge will specify in the TRILL header the RBridge closest to the final destination (called the egress RBridge) to which the frame should be forwarded. Each RBridge on the path between the ingress and the egress RBridge needs to analyze the received frame and change the Outer Ethernet header according to the next hop.

A.2 TRILL header

A structure of the TRILL header is shown at the bottom of the Figure A.1. It contains the following fields:

- Ethertype : TRILL (with value 0x22F3)
- V : Version of the TRILL protocol
- R : Reserved bits for future extensions
- M : Indicates whether a frame is to be delivered to multiple destination end stations via distribution tree specified by the egress RBridge nickname
- OP-length : length of the TRILL header options in 4 byte units.
- Hop count : 6-bit unsigned integer that gets decremented on every hop if greater than 0, otherwise the frame gets dropped
- Egress RBridge Nickname : 16-bit identifier that specifies the Egress RBridge in the case when M==0, otherwise (when M==1) it specifies the distribution tree selected to be used to forward the frame
- Ingress RBridge Nickname : 16-bit identifier that specifies a nickname of the ingress RBridge for TRILL data frames
- Options : exists if OP-length is different from zero. In that case, first octet is used for flag bits among which the first two are:

– CHbH (Critical Hop by Hop) - in the case when this bit is equal to 1, one or more critical hop-by-hop options are present

– CItE (Critical Ingress-to-Egress) - in the case when this bit is equal to 1, one or more critical ingress-to-egress options are present

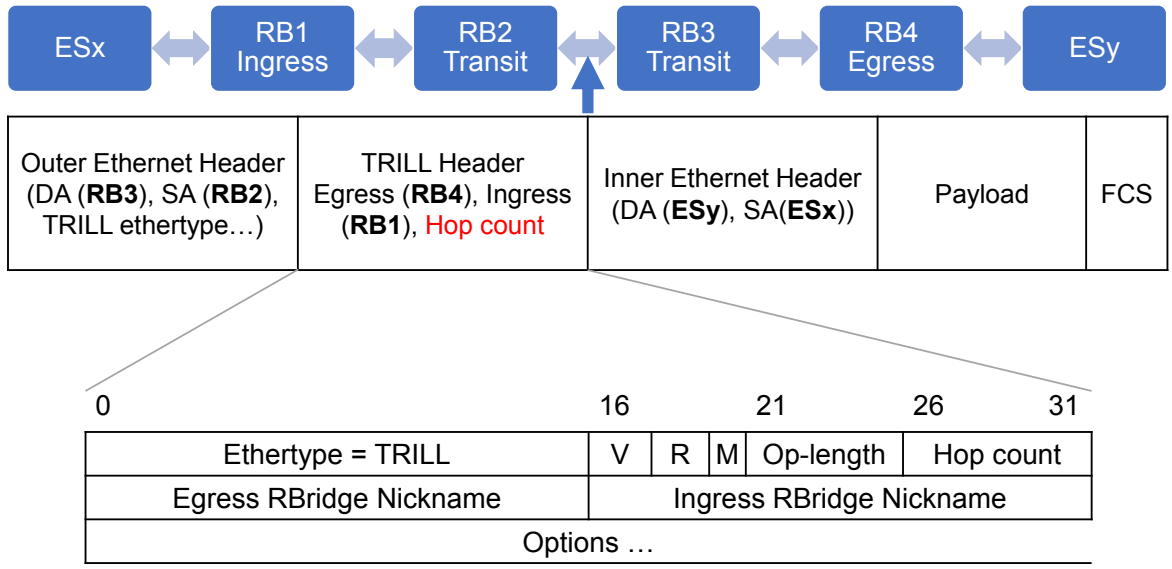


Figure A.1: TRILL header

A.3 TRILL data encapsulation over Ethernet

Figure A.1 shows the communication between end stations ESx and ESy. RBridges (denoted as RBs in Figure A.1) may be interconnected with IEEE 802.3 technology or some other layer 2 technology like PPP (Point-to-Point Protocol). The native frame is generated by ESx, it gets encapsulated by ingress RBridge RB1, and will ultimately be decapsulated by egress RBridge RB4. The first RBridge that a frame encounters in the campus will encapsulate a frame with a TRILL header, as previously described. Furthermore, RBridge will do another encapsulation on the received frame that is link-specific. For instance, it can be encapsulation over PPP (Point-to-Point Protocol), but in this example the encapsulation is done with the outer Ethernet header as the IEEE 802.3 technology is used. From Figure A.1 we can see that 3 headers are present when the frame is transferred from RB2 to RB3. The inner Ethernet header's MAC addresses are set by the original end station and they specify the (final) destination ESy, and the source of the original (inner) frame ESx. The TRILL header specifies the egress RBridge nickname RB4 and the ingress RBridge nickname RB1. The outer Ethernet header's MAC addresses are set by the transmitting RBridge RB2 and they specify the next hop RBridge RB3 and the transmitting RBridge RB2. This header helps to maintain retro-compatibility with conventional customer bridges as they can be located between two RBridges.

At each RBridge hop, the processing of the TRILL header needs to be performed. The outer Ethernet header gets stripped off and it gets replaced by another Ethernet header (in the case Ethernet is used on next hop link). The egress RBridge nickname is used in order to find the next hop destination MAC addresses in the forwarding table. The hop count is decreased in the TRILL header as well. The TRILL frame processing needs to be done on each frame and on each hop. This represents a bottleneck in communication when processing is done on the server. In order to prevent this bottleneck, we decided to offload the data plane processing on the programmable card.

A.4 TRILL LSPs and TRILL-Hellos

A.4.1 TRILL LSPs

TRILL IS-IS LSPs are exchanged between the RBridges in a TRILL campus and they must contain the following information:

1. The IS-IS IDs of neighbors (pseudonodes and RBridges) of the RBridge RBn and costs of the links to each of the neighbors [54]
2. 2-byte nickname value including the 8-bit priority to have that nickname and 16-bit priority of that nickname to become a distribution tree root [54]
3. maximum TRILL header version supported
4. additional information related to distribution tree determination and announcement
5. list of VLAN IDs of VLANs directly connected to RBn for links on which RBn is the appointed forwarder for that VLAN [54]

A.4.2 TRILL-Hello protocol

TRILL-Hellos are usually sent with the same timing as layer 3 IS-IS Hellos as the former protocol is based on the later one. In general, TRILL-Hello protocol is used to [178]:

- determine the RBridge neighbors that have acceptable connectivity in order to report them as part of the topology
- elect a unique Designated RBridge on broadcast (LAN) links
- determine the MTU that allows safe communication with each RBridge neighbor

TRILL-Hello protocol is slightly different from the layer 3 IS-IS LAN Hello protocol. There are a few main modifications. The first one is the limited size of Hello messages as they are not padded like in layer 3 IS-IS LAN Hello protocol. Additionally, DRB election is based solely on the priority along with its MAC address and it not on a two-way connectivity as in layer 3 IS-IS. This means that if RB2 receives a TRILL-Hello from RB1 with higher priority, RB2 defers to RB1 as DRB regardless

of whether RB1 lists RB2 as a neighbor in its TRILL-Hello packet [54]. TRILL-Hello protocol has a separate mechanism for probing and it uses packets of different sizes to see what packet sizes can be forwarded on the link. TRILL-Hello messages must not exceed 1470 octets in length.