Chapter

2

SmartNICs: The Next Leap in Networking

Marcelo Caggiani Luizelli (UNIPAMPA), Francisco Vogt (UNICAMP),
Guilherme Mendes Vieira de Matos (UFSCar), Weverton Cordeiro (UFRGS),
Alberto Egon Schaeffer Filho (UFRGS), Marcos Schwarz (RNP),
Fabio Luciano Verdi (UFSCar), Christian Esteve Rothenberg (UNICAMP)

Abstract

Programmable Network Interface Cards (SmartNICs) have gained increasing momentum due to their flexibility to offload complex networking tasks from the host CPU to a programmable hardware architecture. Despite the performance gains (e.g., lower latency), programming, debugging, and operating SmartNICs pose challenges due to diverse hardware architectures (e.g., System-on-Chip, FPGA, and ASIC), various programming languages, and operation modes. This tutorial aims to shed light on the design principles and operation of modern SmartNICs, covering hardware architectures, programming software ecosystem, performance capabilities and open research challenges in this evolving domain, The tutorial concludes with a hands-on experience involving cutting-edge SmartNICs such as Nvidia BlueField, Mellanox ConnectX, and Netronome NFP.

Resumo

As placas de rede programáveis (SmartNICs) têm recebido crescente visibilidade devido à sua flexibilidade para descarregar tarefas de rede complexas da CPU do host para uma arquitetura de hardware programável. Apesar dos ganhos de desempenho (por exemplo, menor latência), a programação, depuração e operação das SmartNICs apresentam desafios devido a diversas arquiteturas de hardware (por exemplo, System-on-Chip, FPGA e ASIC), várias linguagens de programação e modos de operação. Este tutorial tem como objetivo esclarecer os princípios de projeto e operação de aplicações em SmartNICs modernas, abrangendo arquiteturas de hardware, ecossistema de software de programação, capacidades de desempenho e desafios de pesquisa abertos neste domínio em evolução. O tutorial conclui com uma experiência prática envolvendo SmartNICs atuais, como Nvidia BlueField, Mellanox ConnectX e Netronome NFP.

2.1. Introduction and Motivation

The ability to program the network data plane has reshaped the network operations and management landscape, opening up a multitude of opportunities for delivering custom-made networking solutions [Kianpisheh and Taleb 2023]. By running home-brewed networking solutions within network programmable devices, network operators and practitioners have an opportunity to close the management control loop and instrument per-packet processing decisions, at line rate, in the order of nanoseconds.

Recently, there has been an increasing interest from both academy and industry towards programmable NICs (Network Interface Cards), commonly referred to as SmartNICs[1]. For instance, Nvidia (BlueFlied model), Netronome (Agilio model), Intel (IPU model), AMD (Pensando), Xilinx (Alveo models), and others are already battling for shares of this new market. These devices are gaining attention due to their ability to efficiently offload from the host CPU to the SmartNIC hardware a wide range of intricate networking tasks, as well as tasks unrelated to networking – promising high-performance packet processing while reducing the total cost of ownership. Examples of offload tasks include in-network caching [He et al. 2023] and in-network machine learning [Saquetti et al. 2021, Swamy et al. 2022] – just to name a few examples.

The term SmartNIC appeared first in the 1990s, when NICs were mostly used to connect computers to networks [Ponomarev and Ghose 1998]. They performed basic tasks like packet reception, transmission, and simple processing, such as checksum offloading. However, NIC offloading has since evolved to include more advanced and specialized capabilities that are necessary to meet the demands of modern networking environments. This is due to the increasing need for improved performance, efficiency, and scalability. For example, in the 2000s, NICs were developed with a TCP Offload Engine. Later, in the 2010s, NICs evolved to support virtualization requirements such as VLAN and RSS. Nowadays, NICs (or SmartNICs) support a long list of embedded functions, including advanced features such as RoCE (RDMA over Converged Ethernet), stateful connection tracking, or even storage acceleration [Xilinx 2024].

Despite the constant evolution of NICs, they only become "Smart" as networking vendors add some level of programmability to them. There are different levels of SmartNICs programmability. Some vendors allow rewriting the whole hardware description (e.g., FPGA-based SmartNICs), while others allow offloading only specific networking tasks to computing units (e.g., SoC-based SmartNICs). To support such a level of programmability, SmartNICs often rely on a wide variety of hardware platforms and programming languages (e.g., P4, Micro-C, VHDL/Verilog). Nevertheless, programming, debugging, and operating SmartNICs remain a challenging task.

Differently from programmable switches architectures (e.g., Tofino TNA architecture [Intel 2021]), SmartNICs usually follow a run-to-completion model where network packets are assigned to computing units without preemption [Guo et al. 2023]. Therefore, in multi-core SmartNICs, network packet processing may experience variable latency de-

[1]SmartNICs are also commonly touted as DPU (Data Processing Unit) or IPU (Infrastructure Processing Unit). Despite some attempts – mostly driven by marketing – to provide technical differences between a SmartNIC, DPU, and IPU, a well-accepted understanding is lacking. Therefore, and considering the scope and objectives of this book chapter, we use these terms interchangeably.

pending on the program structure, such as the number of Match/Action tables. That is not the case, for instance, in TNA architectures where resource constraints are a first-order concern, and performance guarantees come for free as long as the program fits inside the device.

To shed light on the design of emerging in-network solutions, this chapter introduces essential principles related to the design and operation of state-of-the-art SmartNICs. We cover and discuss different hardware architectures and the available programming software ecosystem. Then, we dive into the performance limitations of existing SmartNICs and discuss tailored networking applications for them. Finally, we present a hands-on tutorial with selected state-of-the-art SmartNICs (Nvidia BlueField, Mellanox ConnectX, and Netronome NFP).

2.2. Fundamentals of SmartNICs

This section provides a solid background on SmartNICs and on the Portable NIC Architecture (PNA) [(PNA) 2023]. We start with a broad overview of the network packet processing pipeline in existing NICs. Then, we will cover the state-of-the-art SmartNIC architectures: (i) Nvidia Bluefield, (ii) Xilinx Alveo, and (iii) Netronome NFP-4000.

As previously mentioned, NICs have evolved to support network speeds that go beyond 100 Gbps, while incorporating programmable units. The hardware architecture necessary for achieving high-speed network packet processing requires a high degree of parallelism to achieve performance scalability in NIC programs. To address this, current programmable NICs rely on multiple hardware architectures including (i) ASIC (e.g., Netronome NFP); (ii) System-on-Chip (e.g., Nvidia BlueField); and (iii) FPGA (e.g., Xilinx Alveo). Our discussion in this section will focus on the referred architectures, which play a central role in many academic studies [Liu et al. 2019, Min et al. 2021, Schuh et al. 2021, Wei et al. 2023a].

SmartNICs typically employ an alternative processing approach in comparison to an ASIC switch (e.g., Tofino), wherein a packet is directed to a specific processing engine following a run-to-completion model. In this model, a packet is assigned to a processor (or specific hardware) which executes all the instructions required for processing the packet. For example, Nvidia BlueField adopts a “disaggregated RMT” architecture [Chole et al. 2017]. In this design, a group of ASIC packet engines handles header computation and retrieves Match/Action (MA) entries from SRAM via a memory bus. In contrast, Netronome Agilio utilizes a collection of SoC-based CPU cores for packet processing, with corresponding entries situated in a more distant memory hierarchy [Xing et al. 2023]. In the case of multicore SmartNICs, packet latency can vary based on the program structure, including factors such as the number of Match/Action tables and their match types. Additionally, packets following distinct execution paths within the same program may encounter varying levels of latency. Despite that, the run-to-completion model is more flexible in the sequence of actions executed on the packet because a processor is not limited in the sequence of actions. This is not the case, for instance, in an RMT pipeline model (e.g., Tofino TNA architecture [Intel 2021]), where the sequence and number of used stages limit the programability.

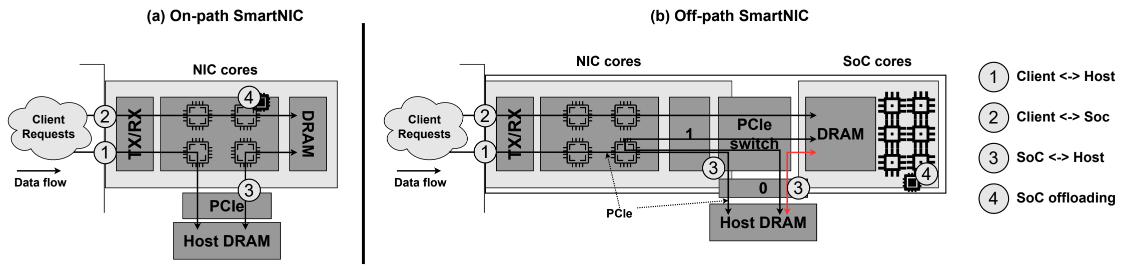


Figure 2.1. Overview of different SmartNIC architectures. Adapted from [Wei et al. 2023b].

2.3. SmartNIC Architectures

In this section, we discuss the main SmartNIC architectures in use on commercial programmable NICs. We start discussing general architectural designs: *on-path* and *off-path*. Then, we deep dive into System-on-Chip (SoC), Field Programmable Gate Arrays (FPGA), Application-Specific Integrated Circuit (ASIC), and hybrid architectures.

2.3.1. On/Off-Path Designs

SmartNICs can be categorized in *on-path* and *off-path* [Wei et al. 2023b]. On-path SmartNICs expose the NIC cores to the system with low-level programmable interfaces. In this design, the offloaded code is on the critical path of the packet processing pipeline and competes for the NIC resources. Therefore, if offloading too much computation, the NIC might suffer performance degradation. Also, NIC's core has direct access to memory subsystems such as DRAM or caching. Figure 2.1 illustrates the On-path design.

In turn, on *off-path* designs, NICs are equipped with additional computing cores and memory in a separate SoC, sitting next to the NIC cores. In this design, the offloaded code is off the critical path of the packet processing pipeline. In this case, the offloaded computation does not affect packet processing performance as long as it does not involve networking communication. Figure 2.1 illustrates the *off-path* design. Observe the computing cores are sitting next to NIC cores, interconnected by an internal PCIe interconnection.

2.3.2. Representative SmartNICs Architectures

2.3.2.1. SoC-based Designs

SoC architecture refers to a microchip that integrates most or all components of a computer or other electronic system into a single integrated circuit. These components typically include a central processing unit (CPU), memory, input/output ports, and various peripheral interfaces such as USB, HDMI, and Ethernet. In the context of SmartNICs, SoC architectures have been used to allow network programmability inside the NIC. Netorking-based SoC is usually composed of multiple CPUs (e.g., ARM cores), integrated with high-speed RAM, and fixed-function accelerators (e.g., encryption and regex engines). On top of the architecture, there is usually a running Operating System that manages NIC resources. In other words, a SoC SmartNIC can be seen as an individual computing node inside the server.

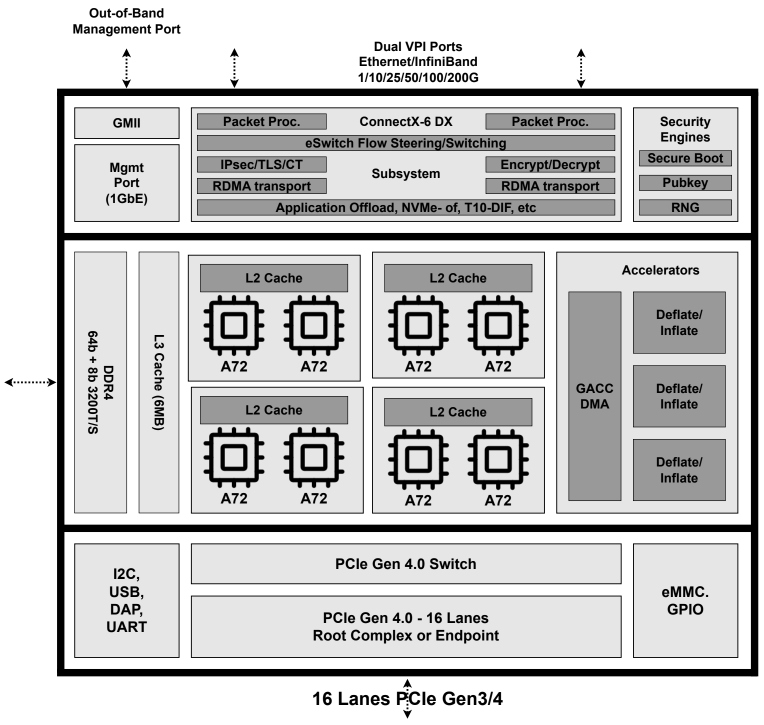


Figure 2.2. Overview of NVIDIA BlueField Architecture. Adapted from [NVIDIA 2024].

One representative SoC-based architecture is the Nvidia BlueField. Figure 2.2 illustrates the 100Gbit Nvidia BlueField-2 architecture. This SmartNIC has eight ARM cores interconnected in a mesh. ARM cores have direct access to DDR4 memory and specific networking accelerators (e.g., regular expression, SHA-2). That allows, for instance, to process packet payloads in hardware using the regular expression engine. One of the fundamental building blocks of this architecture is the eSwitch subsystem. Network packets are received from the Ethernet/Infiniband physical ports. Then, these network packets are processed by the eSwitch Flow Steering subsystem, which can be seen as a programmable switch inside the NIC (e.g., to Open vSwitch[Pfaff et al. 2015]). As such, it allows programming how network packets are steered from/to physical network ports, as well as to steer them to networking applications running on top of the ARM cores, or even send them via PCIe to the host CPU.

As there is an Operating System in the SoC-based SmartNIC, programmers can run any application inside it (ranging from high-performance DPDK-based applications to C-alike sockets). However, to get the best performance out of the architecture, each networking vendors provide a programming suite (e.g. DOCA framework) to better use the available resources (as we later explore in detail in Section 2.7). Therefore, different from a programmable switch (e.g., Tofino and its TNA architecture), SoC-based SmartNIC performance can vary depending on the running networking application. As there is no upper bound on the number of instructions each packet can be submitted to, the line rate (and the latency) can be compromised depending on how intense is the packet processing.

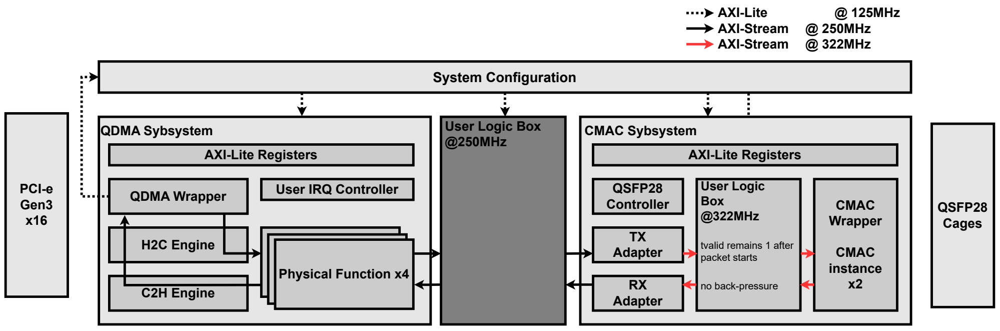


Figure 2.3. Overview of OpenNIC architecture. Adapted from [OpenNIC 2024].

2.3.2.2. FPGA-based Designs

An FPGA is an integrated circuit that allows users to configure it after manufacturing. Unlike conventional integrated circuits, which have fixed functionalities, FPGAs consist of programmable logic blocks and interconnects that can be customized to implement various digital circuits. By loading a configuration file onto the chip, users can specify how the logic blocks should be interconnected and what functions they should perform. This flexibility makes FPGAs suitable for prototyping, fast development, and applications requiring hardware reconfiguration or customization.

In the context of SmartNICs, there are a few vendors that provide FPGA-based SmartNIC (e.g. Xilinx). FPGA SmartNIC provides flexibility to prototype anything directly in hardware. However, the programmer needs to write the whole NIC hardware description (usually using High-Level Synthesis languages), and the operating system drivers. The NIC hardware is usually composed of many components (e.g., DMA subsystems, multiple RX/TX queues, Ethernet MAC subsystems, schedulers, etc). As one can observe, writing an FPGA hardware description might become a hard task to be done. Fortunately, the Open-Source community provides a reference FPGA NIC implementation called the OpenNIC project[2].

The OpenNIC project offers an FPGA-based NIC platform tailored for the open-source community. This platform comprises various components: a NIC shell, a Linux kernel driver, and a DPDK driver. The NIC shell encompasses RTL sources and design files, optimized for deployment on numerous boards featuring UltraScale+ FPGAs. It furnishes a NIC implementation supporting up to four PCI-e physical functions (PFs) and two 100Gbps Ethernet ports. The shell features well-defined data and control interfaces, facilitating seamless integration of user logic. Figure 2.3 illustrates the basic components of OpenNIC project.

2.3.2.3. ASIC-based Designs

ASIC-based SmartNICs utilize customized fixed-function processors (or integrated circuits) to process packets. One representative ASIC-based architecture is the Netronome

[2]`https://github.com/Xilinx/open-nic`

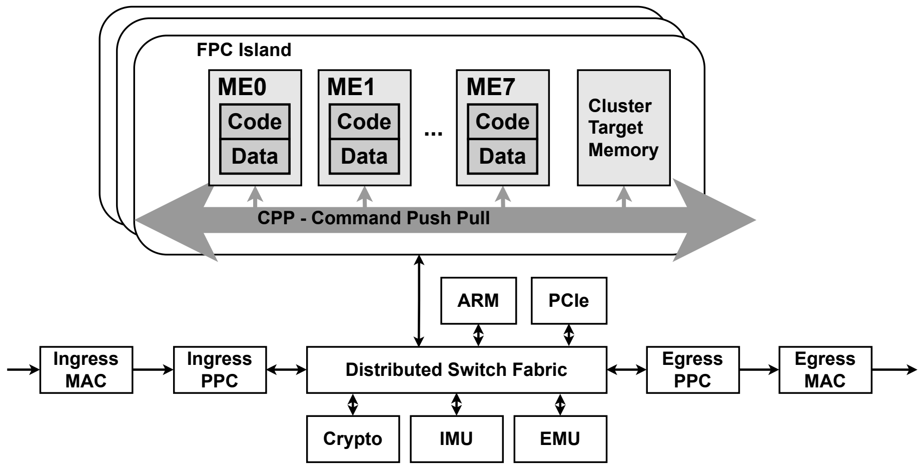


Figure 2.4. Overview of the Netronome architecture. Adapted from [Cannarozzo et al. 2024].

NFP4000. The architecture of the Netronome NFP4000 SmartNIC is depicted in Figure 2.4, with its flow processing cores (FPC) organized into multiple islands.

Each FPC operates as a 32-bit machine, with all internal registers and local memory consisting of 32-bit words. Within each FPC, there are eight Micro Engines (MEs), each functioning as a separate processor with its own instruction store (*code*) and local memory (*data*). Consequently, each ME can execute in parallel with all other MEs. With 8 threads per ME, cooperative multithreading is enabled, allowing at most one thread to execute code from the same program concurrently. Thus, each FPC can accommodate up to 8 parallel threads, running at a frequency of 1.2GHz (one thread per ME). FPCs adhere to a Harvard Architecture, where code and data are stored in separate memories: 4K bytes are shared between threads for data and private memory, while 8K instructions are reserved for the code store.

The local memory within each FPC consists of 32-bit registers shared among all 8 threads, categorized into: (i) general-purpose registers (256 registers of 32 bits each), (ii) transfer registers (512 registers of 32 bits each) for interconnection bus operations, (iii) next-neighbor registers (128 registers of 32 bits each) mainly for communication with neighboring FPCs, and (iv) local memory (1024 registers of 32 bits each), slightly slower than general registers with a 3-cycle access time. When local FPC registers cannot accommodate the required memory, variables are automatically and statically allocated to other in-chip memory hierarchies.

In addition to local memory, FPCs have access to four other types of memory: Cluster Local Scratch (CLS) memory (20-50 cycles), Cluster Target Memory (CTM) (50-100 cycles), Internal Memory (IMEM) (120-250 cycles), and External Memory (EMEM) (150-590 cycles). Each memory type serves specific purposes, ranging from storing frequently used data to managing packet headers and accommodating large shared tables.

Incoming packets from the network are picked up by an FPC thread from the Distributed Switch Fabric and processed accordingly, thereby constituting an on-path Smart-

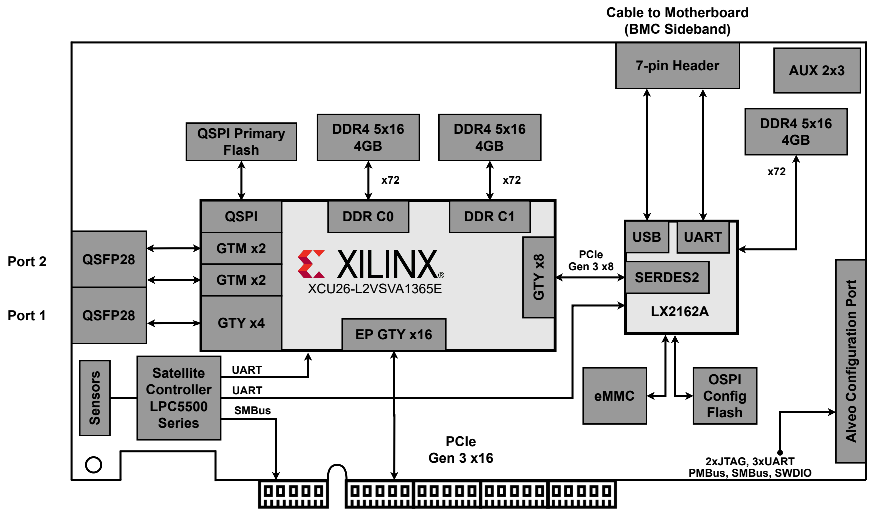


Figure 2.5. Overview of the Xilinx SN1000 architecture. Adapted from [Xilinx 2024].

NIC operation. For instance, the SmartNIC NFP-4000 supports up to 60 FPCs, with each FPC capable of handling up to 8 threads, enabling the device to process up to 480 packets simultaneously.

2.3.3. Hybrid Designs

In addition to the aforementioned SmartNIC architectures, there are still hybrid architectures made of a combination of existing architectures. The most popular combination is the SoC + FPGA architecture. These architectures provide the hardware expressiveness of FPGAs, combined with the flexibility of using ARM cores in the SoC. Note, however, that other architectural combinations also exist such as SoC + ASIC.

Figure 2.5 illustrates the architecture of Xilinx SN1000. As one can observe, the physical ports (QSFP28) are directly interconnected with the FPGA board (XCU26). The FPGA is directly connected to memory multiple subsystems and to the SoC subsystem (LX216A). The SoC in this architecture is composed of 16 ARM cores which also access to private memory subsystem.

2.3.4. Portable NIC Architecture (PNA)

Over the last years, the P4 (Programming Protocol-independent Packet Processors) language [Bosshart et al. 2014a] has been mainly used to program programmable switches. A P4 is an open source, domain-specific programming language for network devices, specifying how data plane devices process packets. The data plane architecture describes the structure and capabilities of the data plane device and exposes architecture-dependent functions to a P4 program [(PNA) 2023] (e.g., hashing functions). The P4 architectural reference (i.e., V1model[3]) reflects the pipeline nature of existent packet processing switches. However, with the emergence of other programmable network devices such as

[3]P4 reference model: `https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4`

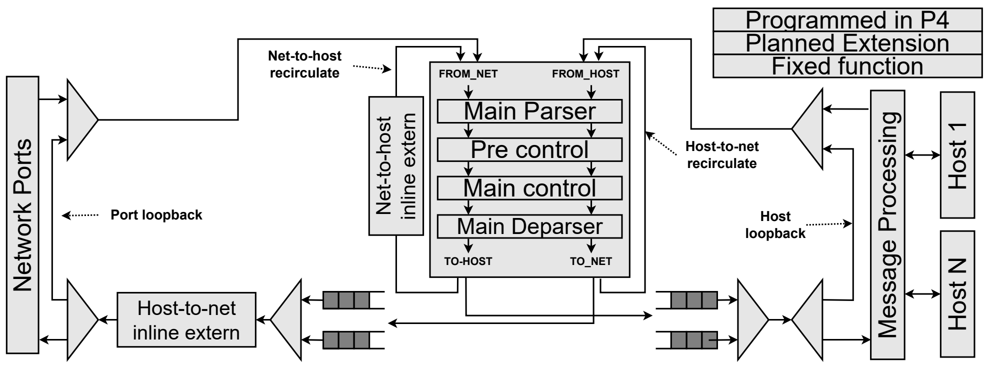


Figure 2.6. Overview of packet paths in the PNA architecture. Adapted from [(PNA) 2023].

SmartNICs, there is a need to adapt that architectural reference to fit existing new requirements/constraints.

In this context, the Portable NIC Architecture is a design framework aimed at encapsulating the fundamental characteristics of a wide array of Network Interface Controller (NIC) devices [(PNA) 2023]. It constitutes a P4 architecture delineating the organization and shared functionalities inherent to NIC devices. PNA consists of two primary elements:

1. A configurable pipeline engineered to accommodate diverse packet pathways traversing between different ports on the device, such as network interfaces or the host system to which it connects.

2. A library of types (e.g., intrinsic and standard metadata) and $P4_{16}$ externs (e.g., counters, meters, and registers).

The PNA Model incorporates four programmable P4 blocks and several fixed-function blocks, depicted in Figure 2.6. The operational characteristics of the programmable blocks are specified using the P4 language [Bosshart et al. 2014b]. Conversely, the network ports, packet queues, and potentially present inline extern blocks are categorized as fixed-function blocks, subject to configuration by the control plane, albeit not intended for programming via P4.

2.3.4.1. PNA Overview

As illustrated in Figure 2.6, network packets can have multiple paths in the context of a NIC. Different from a traditional switch-based P4 architecture (e.g., V1model, TNA, etc), where packets are coming/going to physical ports, in the PNA architecture packets can come from physical network ports, as well as to/from host interfaces. In the context of PNA architecture, these paths are referred to as (i) Net-to-Host, (ii) Host-to-Net, (iii) Net-to-Net, and (iv) Host-to-Host paths.

Packets arriving from a network port go through a MainParser and a PreControl building blocks. The MainParser, similar to other P4 architectures, is intended to parser network packets according to packet headers. For instance, a (Ethernet + IP + TCP) packet is going to be parsed according to the definitions of these three protocols. At the end of the parser process, the header fields of these protocols are stored in data structures that are available throughout the whole packet processing pipeline. Then, in the PreControl block, packets can optionally perform table lookups. Its purpose is to determine whether a packet requires processing by the net-to-host inline extern block. An example of an inline extern block that packets might be submitted to in the Net-to-Host path is the decryption of packet payloads according to the IPsec protocol. The PreControl code might drop the packet if the packet had an IPsec header, but one or more P4 table lookups determined that the packet does not belong to any security association that had been established by the control plane.

The MainControl is typically where the code would be written for processing packets. It transforms headers, updates stateful elements like counters, meters, and registers, and optionally associates additional user-defined metadata with the packet. The MainDeparser serializes the headers back into a packet that can be sent onward. After the MainDeparser, the packet may either: proceed to the message processing part of the NIC, and then typically on to the host system, or turn around and head back towards the network ports. This enables on-NIC processing of port-to-port packets without ever traversing the host system.

While the primary programmable blocks focus on handling individual network packets, typically limited to a single network maximum transmission unit (MTU) in size, the message processing block assumes the responsibility of facilitating the conversion between large messages stored in host memory and network-sized packets for transmission across the network. Moreover, it manages interactions with one or more host operating systems, drivers, and/or message descriptor formats residing in host memory.

For instance, when converting large messages to network packets in the host-to-network direction, the message processing block orchestrates functionalities such as Large Send Offload (LSO), TCP Segmentation Offload (TSO), and Remote Direct Memory Access (RDMA) over Converged Ethernet (RoCE). Conversely, in the network-to-host direction, it aids in features such as Large Receive Offload (LRO) and RoCE.

2.3.4.2. Differences from P4$_{16}$ language

Although the PNA architecture relies on the P4$_{16}$ language definition, some new features are still not defined in the base P4 language. In particular, there are mach-action table properties that are not included in the base P4$_{16}$ language specification.

The *add_on_miss* table property is defined by the PNA. When this property is set to true for a table, P4 developers can specify a default action for the table, which invokes the *add_entry* extern function. This function adds a new entry to the table, where the default action involves calling the *add_entry* function. The newly added entry will possess the identical key that was recently queried. That allows to create, for instance, self-contained data plane applications that do not depend on the control plane. The control plane API would still be able to add, change, and remove entries from the mentioned table. However, entries added through the *add_entry* function operate independently of the control plane software.

Another table property envisaged in the PNA architecture is the *table entry timeout notification*. PNA utilizes the *pna_idle_timeout* parameter to allow a table implementation in sending notifications from the PNA device. These notifications are triggered when a configurable duration has elapsed since an entry was last matched.

2.3.4.3. Existing efforts to make PNA usable

Despite the growing involvement of the Open-Source community to make PNA a standard architecture for SmartNICs, little is still possible to be done using PNA to program SmartNICs. To date, the only P4 backend compiler that adheres to the PNA architecture is the `p4c-dpdk`[4]. The `p4c-dpdk` backend translates the P4-16 programs to DPDK API to configure the DPDK software switch (SWX) pipeline. DPDK introduced the SWX pipeline in the DPDK 20.11 LTS release. Each pipeline is created using a specification file that can either be manually developed or generated using a P4 compiler.

Therefore, one can write P4-based PNA code, compile it to DPDK, and run it either on SmartNICs (e.g., SoC-based ones) or in the host system using DPDK-enabled NIC interfaces using the DPDK pipeline application. For more information, please refer to the official DPDK documentation [DPDK 2024].

2.4. SmartNIC Software & Hardware Ecosystem

In this section, we provide an overview of SmartNICs from the vendor's perspective. We discuss the available SmartNIC models considering the existing architectures, the programming suite available, and existing hardware requirements and specifications.

2.4.1. Brief Historical Perspective

The emergence of the SmartNIC term dates back to the 90s [Ponomarev and Ghose 1998]. However, at that time, NICs were primarily responsible for interfacing a computer with a network. They handled basic functions such as packet reception, transmission, and some basic processing tasks like checksum offloading.

NIC offloading has evolved from basic functionality to more advanced and specialized offloading capabilities, driven by the increasing demands of modern networking environments and the need for improved performance, efficiency, and scalability. For instance, in the 2000's, NICs have been empowered with TCP Offload Engine. Later, in the 2010's, NICs evolved to support virtualization-driven requirements such as VLAN, RSS, etc.

However, it was only after the emergence of Software-Defined Networking (SDN) (and later the P4 language) concepts that SmartNICs started to emerge as a "smart" device. The network programmability fostered by these concepts has driven the development of

[4]`https://github.com/p4lang/p4c/blob/main/backends/dpdk/README.md`

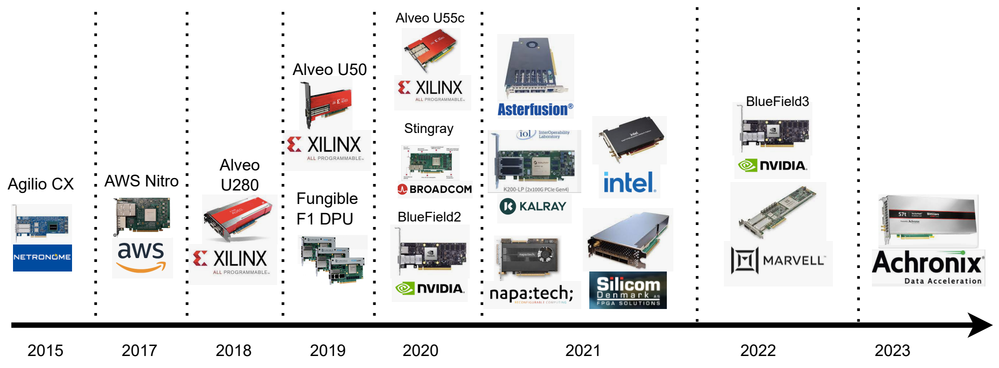


Figure 2.7. SmartNIC vendors timeline.

high-level programmable NICs. Figure 2.7 illustrates SmartNIC vendors over the last year. Note that this is not an exhaustive list and many models are not mentioned.

In 2015, Netronome[5] was the first vendor to announce a SmartNIC model supporting data plane user-level programmability using P4 language. They provided a P4-to-MicroC compiler, allowing the user to write their own NIC firmware with full access to the NIC computing power. Then, in 2017, Cloud players also announced the design of SmartNICs to support the ever-growing demands of their data centers (e.g., AWS Nitro). In 2018, Xilinx announced the Alveo SmartNICs designed also aimed at data center workloads. These cards were one of the first FPGA-based SmartNICs. These cards leverage Xilinx's FPGA technology to accelerate a wide range of compute-intensive tasks. In 2019 and 2020, SoC-based SmartNICs were announced such as the NVIDA BlueField. Following this trend, in 2021, Intel also announced Intel SoC interfaces. More recently, Marvel and Asterfusion also announced powerful SoC-based SmartNICs with up to 36 ARM cores. Table 2.1 summarizes the main SmartNICs by vendor.

As we observe in Table 2.1, SmartNICs can handle a few hundred gigabytes per second. In 2015, Netronome architecture was able to handle up to 40Gbit/s. Today, we see SmartNICs supporting up to 200Gbit/s per physical port. SmartNIC models are evolving together with just approved Ethernet standards. The IEEE P802.3bs Task Force developed the 400 Gigabit Ethernet (400G, 400GbE) and 200 Gigabit Ethernet (200G, 200GbE) standards, which were approved back in 2017. These standards employ technology similar to that of 100 Gigabit Ethernet. Last, in 2024, the IEEE P802.3df Task Force approved the 800 Gigabit Ethernet (800G, 800GbE) standard.

[5]`https://netronome.com/`

Table 2.1. An overview of SmartNIC ecosystem.

Vendor	Model	Release Year	Architecture	Bandwidth	PCIe	Programming Suite	Features
Achronix	Speedster7t FPGA series	2023	FPGA	400 Gbit/s	PCIe 5.0	Achronix Tool Suite (Verilog/VHDL)	Achronix FPGA 692K LUTs
Napatech	NT200A02	2021	FPGA	100Gbit/s	PCIe 3.0	Vivado Suite	Xilinx FPGA UltraScale+
Xilinx	u280	2018	FPGA	100Gbit/s	PCIe 4.0	Vivado Suite	Xilinx FPGA UltraScale+ 1079K LUTS
Xilinx	u55c	2020	FPGA	100Gbit/s	PCIe 4.0	Vivado Suite	Xilinx FPGA UltraScale+ 1079K LUTS
Intel	N5010	-	FPGA	4x 100Gbit/s	PCIe 4.0	Intel Open FPGA Stack (Intel OFS)	Intel Stratix 10 DX 2073K LUTS
Intel	N6000-PL	2021	FPGA	200Gbit/s	PCIe 4.0	-	Intel FPGA Agilex 7 FPGAs F-Series 1437k LUTS
Asterfusion	Helium EC2002P	2021	SoC	200Gbit/s	PCIe 3.0/PCIe 4.0	DPDK/VPP Development Kit	24 ARM 1.8Ghz + Dedicated acellerator + ASIC
Fungible/Microsoft	F1 DPU	2019	SoC	800Gbit/s	4x PCIe 4.0	P4	MIPS-64, 9-stage, dual-issue, 4xSMT, FPU/SIMD unit
Kalray	K200/K200-LP	2021	SoC	100Gbit/s	PCIe 4.0	P4	5 cluster of 16 cores – MPPA – specific cores – 600 to 1.2 Ghz
Marvell	LiquidIO III	2022	SoC	2x 50Gbit/s	PCIe 4.0	DPDK, VPP, SPDK	Up to 36 cores ARM V8 at 2.2GHz
NVDIA	BlueField-2	2020	SoC	2x 100Gbit/s	PCIe 4.0	DOCA framework, DPDK	8 Armv8 A72 cores
NVIDIA	BlueField-3	2022	SoC	400Gbit/s	PCIe 5.0	DOCA framework, DPDK	16 Armv8.2+ A78 Hercules cores
Broadcom	Stingray	2020	SoC	100Gbit/s	PCIe 4.0	DPDK	cluster of four 3 GHz dual-core Arm v8 A72 complexes
Intel	Intel FPGA IPU F2000X-PL Platform	2021	SoC + FPGA	200Gbit/s	4x PCIe 4.0	IPDK, SPDK P4 Programmable	Intel Agilex 7 FPGA F-Series 2300k LUTS + 8-core Intel Xeon-D SoC
Xilinx	SN1000	2021	SoC+FPGA	200Gbit/s	PCIe 4.0	Vivado, Vitis, P4, DPDK	AMD XCU26 FPGA Ultra-Scale+, 16-core Arm
Netronome	Agilio CX	2015	ASIC	40Gbit/s	PCIe 3.0	P4, Micro-C / Agilio Software	48/60 NFP-4000 flow processor

2.4.2. Programming SmartNICs

Designing and implementing SmartNIC programs might take time and effort, depending on the vendor and available architecture. Despite increasing interest in PNA architecture and P4 language, they are still very restricted and are not widely available. For this reason, each vendor/model takes advantage of its development framework. Next, we overview the main programming frameworks available from the leading vendors: Netronome, Xilinx, and NVIDIA. We later deep dive into the NVIDIA framework as it is used in Section 2.7 in a step-by-step, hands-on programming tutorial.

2.4.3. ASIC-based SmartNIC: Netronome Programming Framework

The Netronome was the first vendor to support P4 language as the primary way to program the SmartNIC data plane. Netronome supports $P4_{14}$ and $P4_{16}$, however much of the development framework is still limited to $P4_{14}$. That includes, for instance, the Netronome simulation engine.

The P4 development in the Netronome board follows a similar architecture to the v1model reference. This allows programmers to translate P4 code seamlessly into the SmartNIC architecture. Besides programming the SmartNIC using P4 language, Netronome allows users to write low-level Micro-C code. Micro-C can be written as P4 externs or as a standalone data plane application. This way, users can get the best out of the Netronome hardware architecture. For instance, Netronome P4 compiler assumes the code is identically running in all NFP cores. Furthermore, it applies simple heuristic procedures to utilize the memory hierarchy (e.g., NFP caches, DRAM, etc). Therefore, the P4 programming might end up with data plane misbehavior for applications that utilize external memory extensively. In those situations, it is recommended to write Micro-C code instead of relying on P4 translations.

The Netronome SmartNIC is backed up by a Linux Run Time Environment (RTE) service that allows flexible interaction with the SmartNIC application from the operating system perspective. That allows, for example, populating Match-Action tables or creating more refined Control Plane applications. Unfortunately, OpenFlow or P4Runtime are not supported by default.

In addition to allowing the data plane firmware to be written from scratch, the Netronome also provides Linux drivers with native support for eBPF/XDP. Therefore, it also allows hardware programability when offloading eBPF/XDP instructions to be executed by the SmartNIC.

2.4.4. FPGA-based SmartNIC: Xilinx Programming Framework

Xilinx Vitis represents a pioneering advancement in development environments, spearheaded by Xilinx to redefine the dynamics of embedded software and hardware platform development. Within its unified interface, Vitis seamlessly converges software and hardware development tasks, accommodating a range of programming languages, including C, C++, OpenCL, and the P4 language. This comprehensive integration allows developers to leverage the unique capabilities of the P4 language, which is particularly advantageous for programmable data planes in networking applications, thereby enhancing both flexibility and performance in their projects.

Central to Vitis is its holistic suite of features, meticulously designed to simplify and elevate the development process. Developers can succinctly articulate algorithmic descriptions through high-level abstractions, while pre-built acceleration libraries cater to common tasks such as image processing and machine learning, optimizing performance and efficiency. The environment's inherent versatility extends to seamless integration with Vivado, Xilinx's acclaimed FPGA synthesis and implementation tool, ensuring a seamless transition from algorithmic formulation to hardware realization. Such synergies empower developers to efficiently map their applications onto Xilinx hardware platforms, accelerating innovation across diverse domains, including artificial intelligence, high-performance computing, automotive, aerospace, and telecommunications.

Moreover, Xilinx offers flexible licensing options for Vitis, providing developers with tailored access to its comprehensive suite of tools and features. Whether through perpetual licenses or subscription-based models, developers can choose the licensing option that best aligns with their project requirements and budget constraints. With Vitis and its adaptable licensing structures, developers are equipped to confidently navigate the complexities of modern development landscapes, forging new frontiers in technology and driving forward the boundaries of possibility.

2.4.5. SoC-based SmartNIC: Nvidia Programming Framework

NVIDIA DOCA brings together a wide range of powerful APIs, libraries, and frameworks for programming and accelerating modern data center infrastructures. DOCA is a consistent and essential resource across all existing and future generations of BlueField DPUs. DOCA offers a rich set of APIs that allow interactions with the BlueField computing units. These include DOCA Flow, DOCA Core, DOCA RDMA, and DOCA GPUNetIO, among others. For a complete list of APIs, please consult the official documentation[6]. In this chapter, we overview the DOCA Flow API, as this is used later for a hands-on tutorial.

DOCA Flow is the most fundamental API available, as it enables the creation of generic packet processing pipes in hardware. The DOCA Flow library provides an API for constructing a set of pipes, each consisting of match criteria, monitoring, and a set of actions. Pipes can be chained so that after a pipe-defined action is executed, the packet may proceed to another pipe. A pipe is a template that defines packet processing without adding any specific hardware rule. It comprises four elements: Match, Monitor, Actions, and Forward.

Figure 2.8 illustrates a pipeline implementation in DOCA Flow. On using DOCA Flow, it is easy to develop hardware-accelerated applications that have a match on up to two layers of packets: (i) MAC/VLAN/ETHERTYPE, (ii) IPv4/IPv6, (iii) TCP/UDP/ICMP, (iv) GRE/VXLAN/GTP-U, or (v) packet metadata. The execution pipe can also have monitoring actions such as counters and policers. Last, the pipe also has a forwarding target: software application (RSS to subset of queues), physical/virtual port, another pipe, and Drop packets.

DOCA Flow pipes offer a versatile framework with a user-defined set of matches

[6] `https://docs.nvidia.com/doca/`

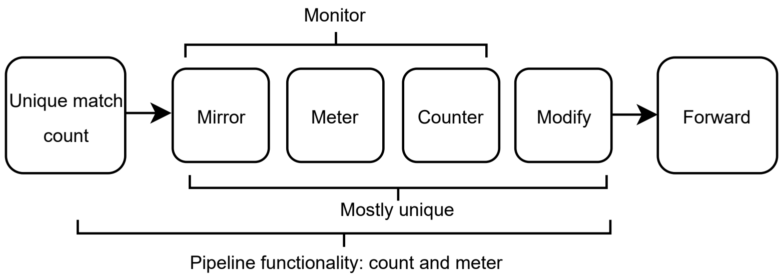


Figure 2.8. DOCA Flow Pipeline. Adapted from NVIDIA DOCA documentation [NVIDIA 2024].

parser and actions, allowing for precise control over packet processing. These pipes can dynamically create or destroy, adapting to changing network demands seamlessly. Leveraging specialized hardware acceleration, packet processing within these pipes achieves optimal efficiency. Each flow pipe contains specific entries tailored to accelerate packet handling, ensuring high-performance throughput. In cases where packets fail to match any hardware entries, Arm cores step in for exception handling, providing a robust mechanism to address diverse scenarios. Following exception handling, packets are seamlessly re-injected back into the hardware pipeline, maintaining the flow's integrity and efficiency.

2.5. Performance Benchmarking

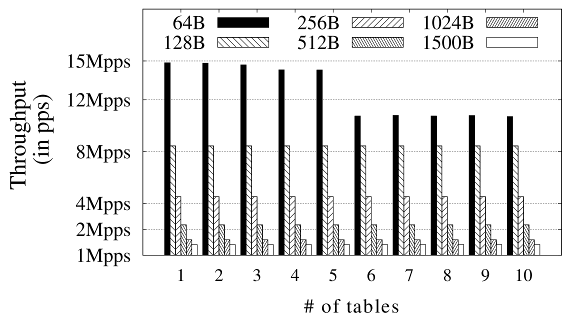
This section focuses on the details of performance benchmarking designed specifically for certain models of SmartNICs. We examine important concepts, methodologies, and key considerations from recent academic studies that have benchmarked some SmartNIC models.

It is worth noting that, despite the growing popularity of SmartNICs, there have been relatively few efforts to benchmark these hardware devices. Furthermore, benchmarking is usually conducted individually and does not involve comparing multiple models or architectures. Nevertheless, we hope that this summary of existing benchmarks can provide an overview of the current performance of various SmartNIC models. In the following, we provide information about some of the performance evaluations available in the literature. These evaluations cover relevant aspects of the current performance and bottlenecks of SmartNICs, including the Netronome NFP4000 SmartNIC and the SmartNICs from the Mellanox ConnectX family.

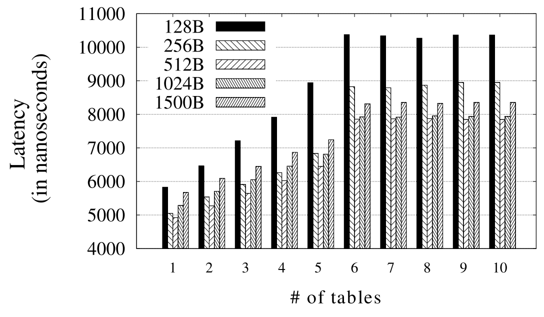
2.5.1. Performance Evaluation of ASIC Netronome NFP4000 SmartNIC

As previously mentioned, the Netronome NFP4000 represented a pioneering advancement in the domain of SmartNICs, characterized by its sophisticated architecture and versatile programmability. In the study of Viegas et al. [Viegas et al. 2021], they provide detailed benchmarking of the performance of P4 instructions running on the NFP4000. To do this, they evaluate the cost (in terms of latency and throughput) of the main operations available in P4, in addition to varying SmartNIC parameters, such as the use of Micro Engines (or processing units). Their evaluation covered aspects such as tables, register operations, recirculations, hash functions, ingress and egress pipelines, and packet sizes. Next, we summarize their SmartNIC evaluation for table accesses, register operations, and recirculations, as we consider the main operations in current P4 programs.

First, we discuss the results for Match-Action table accesses in P4 programs. In the context of SmartNICs, this evaluation is important because, unlike conventional forwarding devices that utilize Match-Action tables solely for routing purposes, the P4 language allows for more diverse applications of this construction. The authors aim to analyze how using multiple match-action tables at different pipeline stages affects performance. In the experiments, they vary the number of tables in their P4 programs and ensure sequential matching on all tables for every packet. Upon packet matching, an action is invoked to read a single 32-bit data from the table and store it in a metadata structure. Packet size and the number of tables per pipeline (ingress or egress) also vary. Figure 2.9 illustrates the measured throughput 2.9(a) and latency 2.9(b). Throughput behavior shows that for small packets, the throughput remains nearly constant for up to 5 Match-Action tables before experiencing a sharp drop. The Netronome architecture limits a P4 program to 5 tables in each pipeline. Interestingly, the ingress pipeline consistently demonstrates faster memory allocation, even when tables are defined only in the egress pipeline.



(a) Measured throughput cost.

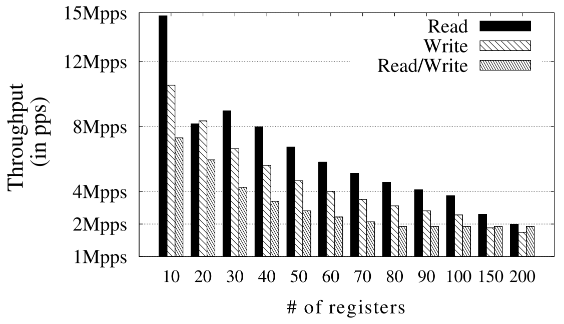


(b) Measured latency cost.

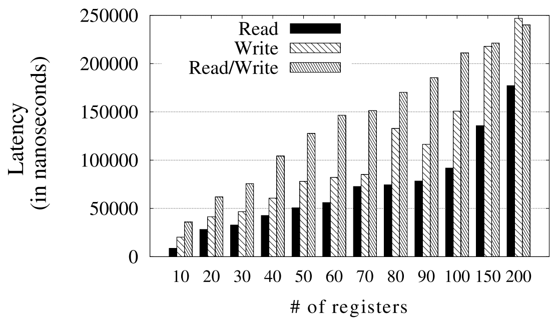
Figure 2.9. Table access costs in the P4 pipeline. Figure from: [Viegas et al. 2021]

To evaluate the memory access costs (register access in P4 code), the authors examine the cost associated with executing multiple register operations within the same P4 pipeline. Register operations serve as fundamental components in modern P4 applications. Throughout the experiments, they varied the number of register operations conducted consecutively by the P4 program, ranging from 10 to 200 register operations, keeping the register width 32-bit. The authors specifically investigate the placement of registers within the ingress pipeline, categorizing them as read-only, write-only, or read-and-write operations. Figures 2.10(a) and 2.10(b) show throughput and latency, respectively. With increasing registers and P4 instructions, significant performance degradation occurs. Line rate sustains only with 10 registers for reading operations, but even with 10 registers, writing operations experience a 30% bandwidth drop (50% for read & write). Throughput linearly decreases by up to 87% (i.e., 2 Mpps) with 200 registers. Latency increases linearly with operation count per packet, from acceptable levels (e.g., 10 registers: 8650ns reading) to higher levels (e.g., 50 registers: 0.12 milliseconds).

Finally, we overview their evaluation of the implications of packet recirculation



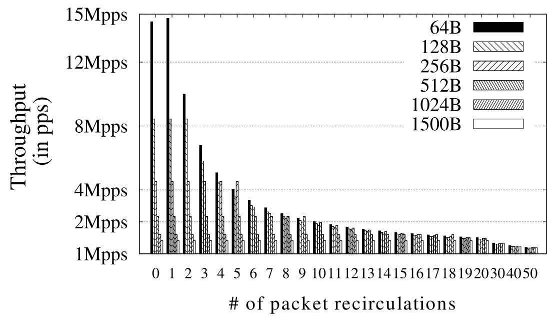
(a) Measured throughput cost.



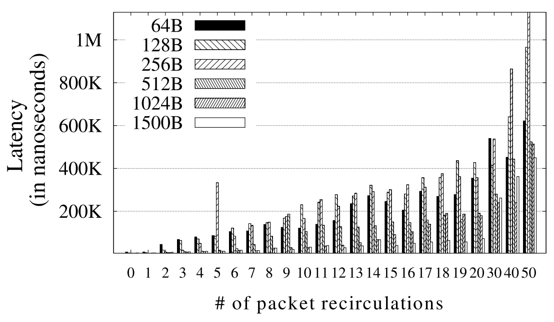
(b) Measured latency cost.

Figure 2.10. Register operation costs in the P4 pipeline. Figure from: [Viegas et al. 2021]

within the P4 pipeline. Given P4's lack of support for iteration-based structures, packet recirculation is a workaround to emulate loop-based functionalities. Packet recirculation involves sending a packet back to the ingress pipeline post-processing, mimicking a loop-based structure. Throughout the experiments, they vary the number of packet recirculations per packet, ranging from 0 to 50, alongside altering the packet size from 64B to 1500B. The focus lies on forwarding network traffic from physical interfaces. Figure 2.11 illustrates the measured throughput 2.11(a) and latency 2.11(b). Throughput behavior shows a super-linear decrease with an increase in packet recirculations. Fewer recirculations sustain a line rate for small packets, while larger packets maintain a line rate even with multiple recirculations. As packets recirculate, more are pushed into the data plane, causing enqueueing and eventual drop, reducing throughput. Observed latency also increases notably as packets recirculate. Even for large packets, per-packet latency doubles with just three recirculations, with sharper increases as the number of recirculations rises, especially for small packets.



(a) Measured throughput cost.



(b) Measured latency cost.

Figure 2.11. Packet recirculation costs in the P4 pipeline. Figure from: [Viegas et al. 2021]

2.5.2. Performance Evaluation of ASIC/SoC Mellanox SmartNIC

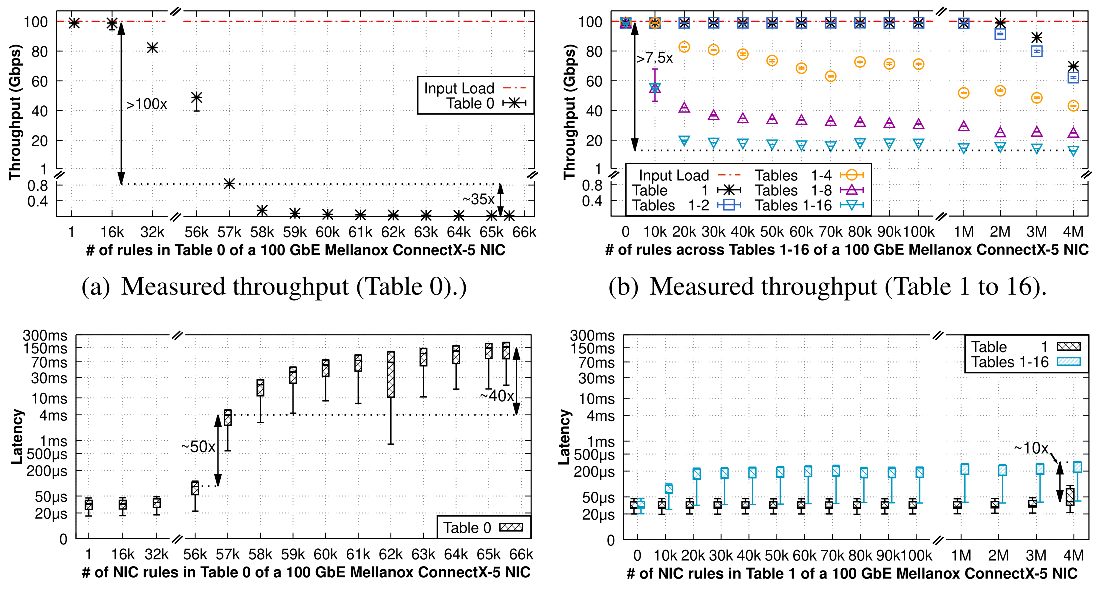
The NVIDIA Mellanox ConnectX SmartNICs offer unparalleled networking performance and processing capabilities, operating at link speeds of up to 100 Gbps and beyond. Other vendors' models have certain limitations such as lower link speeds and limited flow rule capacities. For instance, the upcoming Intel E810 series still needs to catch up when compared to the ConnectX NICs. This demonstrates the superior scalability and performance capabilities of Mellanox SmartNICs. Next, we present a performance evaluation of some ConnextX SmartNICs, providing valuable insights into their performance evolution and highlighting the significance of ConnectX SmartNICs in modern data center environments.

The evaluation carried out by Katsikas et al. [Katsikas et al. 2021] delves into the architectural intricacies and differences among the NVIDIA Mellanox NICs, emphasizing their robust capabilities and innovative design features. The authors evaluate the ConnextX-4, ConnextX-5, ConnextX-6 and BlueField SmartNIC. Leveraging a PCIe 3.0 x16 bus interface, except for the ConnectX-6 adapter, which utilizes two PCIe 3.0 x16 slots, these SmartNICs ensure optimal connectivity and throughput to the server's CPU. Furthermore, including an 8-core ARM processor in the BlueField-2 (and a 16-core ARM processor in BlueField-3) NIC extends its capabilities for in-NIC traffic processing, enhancing overall network performance and efficiency. Notably, Mellanox NICs boast a root table with ample space for rule entries and high-performance exact-match tables that facilitate efficient packet classification and offload tasks from the CPU. With the flexibility to accommodate many rules limited only by the host's available memory, Mellanox SmartNICs emerge as indispensable components in modern server architectures, offering unmatched scalability, performance, and flexibility for demanding networking environments.

In their evaluation, the authors cover aspects such as the cost of the number of table rules pre-installed, the impact of batch and rate-based updates, the impact of inserting new rules, and the impact of in-memory or out-memory updates. In our summary, we only present the cost of the number of rules pre-installed and the cost of rate-based updates, as they are the main table operations. Their experiments use a single-core forwarding network function on a Device Under Test (DUT). The DUT's NIC routes incoming frames to this network function based on flow rules stored in the NIC. These rules are in either the default "root" flow table or non-root tables. The results presented an overview of the ConnectX-5 NIC results, but the observed trends apply to other NICs tested.

Figure 2.12 illustrates how the performance of packet classification varies with the number of rules installed on different types of tables in the NVIDIA Mellanox ConnectX-5 NIC. When rules are uniformly distributed across non-root tables, throughput remains stable even with many entries. However, performance dramatically declines when the root table (Table 0) reaches over 85% occupancy, rendering the last 15% of memory practically unusable. Despite lower input loads, throughput decreases significantly, indicating a design issue with the root table rather than an excessive load. Additionally, while non-root tables exhibit faster throughput and lower latency, spreading rules across more tables leads to performance degradation, highlighting trade-offs in table distribution strategies.

Next, the authors evaluate the cost of updating rules in a rate-based way. Periodic rule installations from a single core are typical in systems like NATs and Layer 4 load balancers. Figure 2.13(a) and Figure 2.13(b) illustrate forwarding network function throughput during simultaneous rule insertions into the NIC classifier. When insertions originate from a different core, throughput remains stable. However, using the same core as the forwarding network function results in a notable performance drop, with throughput



(a) Measured throughput (Table 0).)

(b) Measured throughput (Table 1 to 16).

(c) Measured latency (Table 0).

(d) Measured latency (Table 1 to 16).

Figure 2.12. Throughput and latency costs for a different number of pre-installed table rules. Figure from: [Katsikas et al. 2021]

decreasing by approximately 70 Gbps for 10K and 500K rule insertions per second in Table 0 and Table 1, respectively. This highlights a bottleneck in the NIC's standard API for updating the forwarding table. While using a different core for rule installation helps, it requires costly inter-core communication and consumes significant CPU resources, such as 100% of a core for several hundreds of milliseconds to install 500K rules. Latency increases by more than 2x for Table 0 and 82% for Table 1 when rule insertions occur from the same core, indicating significant performance degradation due to interference in the NIC data plane.

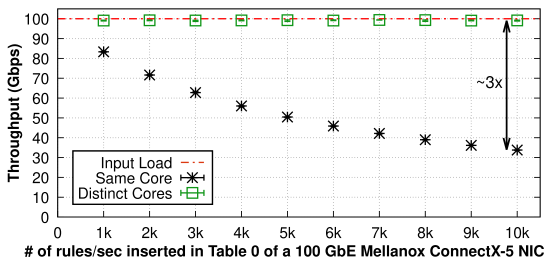
2.6. Related Work

In this section, we describe recent works that used the SmartNICs presented in the previous sections to optimize or accelerate network functions. To do this, we initially present Table 2.2, which contains works published in the last four years (i.e., within 2020-2023) in some of the high-impact conferences that used SmartNICs in their solutions or experiments. In our search, we consider the ACM SIGCOMM (ACM Special Interest Group on Data Communication)[7], USENIX NSDI (Symposium on Networked Systems Design and Implementation)[8], and the ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)[9] conferences, and the table presents the title of the work, conference and year, model, and architecture of the SmartNIC used. Please note that this is not an exhaustive list of all the works published in these conferences. Still, we aim to provide a comprehensive overview of the most relevant recent research studies that highlight the relevance of SmartNIC architectures in high-impact research. There

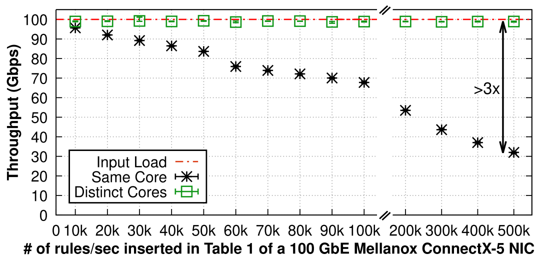
[7]https://www.sigcomm.org/events/sigcomm-conference

[8]https://www.usenix.org/conference/nsdi23

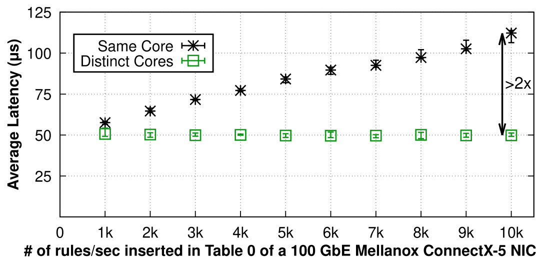
[9]https://conferences.sigcomm.org/co-next/2023/



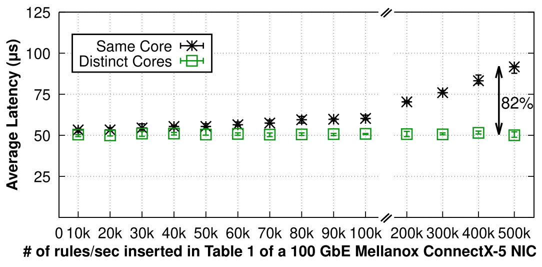
(a) Measured throughput (Table 0)).



(b) Measured throughput (Table 1).



(c) Measured latency (Table 0).



(d) Measured latency (Table 1).

Figure 2.13. Throughput and latency costs for rate-based updates in table rules. Figure from: [Katsikas et al. 2021]

may be other works related to SmartNICs in other high-impact conferences and journals that we may have missed.

Table 2.2. An overview of recent scientific publications featuring SmartNICs.

Year	Conference	Paper	SmartNIC model	Architecture
2023	SIGCOMM	Unleashing SmartNIC Packet Processing Performance in P4	Nvidia BlueField2 (2 ports×100Gbps).	SoC
2023	SIGCOMM	Lightning: A Reconfigurable Photonic-Electronic SmartNIC for Fast and Energy-Efficient Inference	Nvidia A100X DPU	SoC
2023	SIGCOMM	LEED: A Low-Power, Fast Persistent Key-Value Store on	Mellanox ConnectX-5 NIC	ASIC
2023	SIGCOMM	ClickINC: In-network Computing as a Service in Heterogeneous Programmable Data-center Networks	Netronome smartNIC	ASIC
2023	SIGCOMM	Direct Telemetry Access	Mellanox Bluefield-2 DPU	SoC
2023	SIGCOMM	DBO: Fairness for Cloud-Hosted Financial Exchanges	Nvidia ConnectX-5 NIC	ASIC
2023	SIGCOMM	BMW Tree: Large-scale, High-throughput and Modular PIFO Implementation using Balanced Multi-Way Sorting Tree	Xilinx Alveo U200	FPGA
2023	SIGCOMM	NeoBFT: Accelerating Byzantine Fault Tolerance Using Authenticated In-Network Ordering	Xilinx Alveo U50 FPGA	FPGA
2023	CONEXT	SPADA: A Sparse Approximate Data Structure representation for data plane per-flow monitoring	Xilinx Alveo U280	FPGA
2023	NSDI	SRNIC: A Scalable Architecture for RDMA NICs	Xilinx FPGA	FPGA
2023	NSDI	Rearchitecting the TCP Stack for I/O-Offloaded Content Delivery	Mellanox BlueField-2	SoC
2023	NSDI	Waverunner: An Elegant Approach to Hardware Acceleration of State Machine Replication	Xilinx U280 FPGA	FPGA
2023	NSDI	LASH: Towards a High-performance Hardware Acceleration Architecture for Cross-silo Federated Learning	Xilinx VU13P FPGA	FPGA
2023	NSDI	ExoPlane: An Operating System for On-Rack Switch Resource Augmentation	Netronome Agilio 40 Gbps smart NICs	ASIC
2023	NSDI	RingLeader: Efficiently Offloading Intra-Server Orchestration to NICs	100G Alveo U280 Data Center Accelerator Card	FPGA
2022	SIGCOMM	Predictable vFabric on Informative Data Plane	Xilinx Alveo U200 card,	FPGA
2022	SIGCOMM	Implementing ChaCha Based Crypto Primitives on Programmable SmartNICs	Netronome Agilio CX 40 Gbit/s	ASIC
2022	CONEXT	PipeDevice: A Hardware-Software Co-Design Approach to Intra-Host Container Communication	Intel Arria 10 FPGA	FPGA
2022	NSDI	FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism	Netronome Agilio CX40 40 Gbps	ASIC
2022	NSDI	Re-architecting Traffic Analysis with Neural Network Interface Cards	Netronome Agilio CX, with an NFP4000	ASIC
2022	NSDI	Gearbox: A Hierarchical Packet Scheduler for Approximate Weighted Fair Queuing	Xilinx Alveo U250	FPGA
2022	NSDI	Enabling In-situ Programmability in Network Data Plane: From Architecture to Language	Xilinx Alveo U280	FPGA
2022	NSDI	An edge-queued datagram service for all datacenter traffic	Mellanox BlueField 2)	SoC
2022	NSDI	Buffer-based End-to-end Request Event Monitoring in the Cloud	Broadcom PS225 SmartNICs	SoC
2022	NSDI	Yeti: Stateless and Generalized Multicast Forwardin	NetFPGA SUME [	FPGA
2022	NSDI	solation Mechanisms for High-Speed Packet-Processing Pipeline	Xilinx Alveo U250 board	FPGA
2022	NSDI	Tiara: A Scalable and Efficient Hardware Acceleration Architecture for Stateful Layer-4 Load Balancing	Xilinx FPGA-based SmartNIC	FPGA
2021	SIGCOMM	Gimbal: Enabling Multi-tenant Storage Disaggregation on SmartNIC JBOFs	Mellanox ConnectX-5 NIC	ASIC
2021	SIGCOMM	A Cloud-Scale Per-Flow Backpressure System via FPGA-Based Heavy Hitter Detection	Xilinx's vu9p	FPGA
2021	SIGCOMM	CAMES: Enabling Centralized Automotive Embedded Systems with Time-Sensitive Network	FPGA	FPGA
2021	SIGCOMM	NanoTransport: A Low-Latency, Programmable Transport Layer for NICs	Chisel FPGA	FPGA
2021	SIGCOMM	CocoSketch: High-Performance Sketch-based Measurement over Arbitrary Partial Key Query	Xilinx Alveo u280	FPGA
2021	SIGCOMM	SiP-ML: High-Bandwidth Optical Network Interconnects for Machine Learning Training	Stratix V FPGAs	FPGA
2021	SIGCOMM	Concordia: Teaching the 5G vRAN to Share Compute	FPGA (Terasic DE5-Net) f	FPGA
2021	SIGCOMM	wo beams are better than one: Towards Reliable and High Throughput mmWave Links	Artix-7 FPGA	FPGA
2021	SIGCOMM	1Pipe: Scalable Total Order Communication in Data Center Networks	Mellanox ConnectX-4	ASIC
2021	NSDI	BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing	Netronome Agilio CX 2x40GbE	ASIC
2021	NSDI	Flightplan: Dataplane Disaggregation and Placement for P4 Programs	Xilinx ZCU102 FPGA	FPGA
2021	NSDI	ATP: In-network Aggregation for Multi-tenant Learning	Mellanox ConnectX-5	ASIC
2021	NSDI	CodedBulk: Inter-Datacenter Bulk Transfers using Network Coding	Xilinx Virtex-7 XC7VX690T FPGA	FPGA
2021	NSDI	Verification and Redesign of OFDM Backscatter	Microsemi AGLN250 low power FPGA	FPGA
2021	NSDI	Simplifying Backscatter Deployment: Full-Duplex LoRa Backscatter	AGLN250 Igloo Nano FPGA	FPGA
2020	SIGCOMM	An Artifact Evaluation of NDP	NetFPGA	FPGA
2020	SIGCOMM	Using Deep Programmability to Put Network Owners in Control	Xilinx SmartNIC	SoC
2020	SIGCOMM	Defending lightweight virtual switches from cross-app poisoning attacks with vifc	netFPGA	FPGA
2020	SIGCOMM	Optimized Tracing of iCF-enabled Programmable Data Planes	netFPGA	FPGA
2020	CONEXT	DeepMatch: Practical Deep Packet Inspection in the Data Plane using Network Processor	Netronome	ASIC
2020	NSDI	NetTLP: A Development Platform for PCIe devices in Software Interacting with Hardware	Xilinx Kintex 7 FPGA	FPGA
2020	NSDI	TinySDR: Low-Power SDR Platform for Over-the-Air Programmable IoT Testbeds	FPGA (not specified)	FPGA
2020	NSDI	Enabling Programmable Transport Protocols in High-Speed NICs	Kintex Ultrascale+ XCKU15P FPGA	FPGA
2020	NSDI	AccelTCP: Accelerating Network Applications with Stateful TCP Offloading	Netronome Agilio LX 40GbE NIC	ASIC
2020	NSDI	VMscatter: A Versatile MIMO Backscatter	FPGA Microsemi Igloo Nano AGLN250	FPGA

Next, in addition to the general overview provided in Table 2.2, we also discuss recent works that use SmartNICs in their solutions in more detail. We divided this description into subsections based on SmartNIC architectures and discussed three works for each architecture.

2.6.1. FPGA-based SmartNICs

The increasing reliance of data center applications on proxies that shift application layer processing into the network motivated the emergence of new packet dispatch strategies. However, prior work has shown that this comes with a performance and resource cost. Wang et al. [Wang et al. 2023] aim to address this challenge by investigating the possibility of offloading Layer 7 (L7) processing to hardware, specifically FPGA SmartNICs. They focus on L7 dispatch, which involves analyzing application requests and distributing them to target service processes. Unlike L3/4 processing, L7 logic operates on application data spanning multiple packets, making offloading more complex due to variable-length fields and packet reassembly. The authors introduce QingNiao, a solution comprising a co-designed protocol, application interface, and hardware design, aiming to offload L7 dispatch efficiently. Prototyped on an FPGA integrated with a 100Gbps Corundum NIC, QingNiao demonstrates programmability and performance gains over software-based L7 dispatch. Results indicate throughput improvements of 7.5x to 8x and latency reductions of 72.5% to 74% compared to state-of-the-art software solutions, showcasing the viability of hardware offloading for L7 processing. The authors have made their software and hardware designs open-source to facilitate further research in L7 processing offload.

Yao et al. [Yao et al. 2023] address challenges in network scheduling by introducing the Balanced Multi-Way sorting Tree (BMW-Tree). This novel data structure aims to improve the efficiency and scalability of scheduling algorithms, particularly in modern data centers with high flow volumes. The BMW Tree enables the realization of the Push-In-First-Out (PIFO) model, which is crucial for packet prioritization. The paper presents two hardware designs, register-based (R-BMW) and RPU-driven (RPU-BMW), leveraging the BMW-Tree concept. These designs offer high throughput and scalability, addressing the limitations of traditional implementations. Evaluation of the proposed hardware designs using Verilog targeting a Xilinx Alveo U200 Data Center Accelerator Card demonstrates their significant performance improvements over traditional PIFO implementations. For instance, an 11-level 2-way R-BMW achieves a throughput of 192 Mpps (million packets per second) for 4k flows, while an 8-level 4-way RPU-BMW supports 87k flows at 93 MHz.

In turn, Zhang et al. [Zhang et al. 2023] explore the challenges of federated learning (FL) and the cryptographic techniques used to secure cross-silo FL operations. It identifies nine common cryptographic operations and highlights the performance degradation. The study proposes the FLASH architecture for hardware acceleration, focusing on FPGA and ASIC implementations. Evaluation of FLASH demonstrates significant performance improvements over CPU and GPU implementations across cryptographic operations and FL applications. Specifically, FLASH outperforms CPU and GPU by 10.4x to 14.0x and 1.4x to 3.4x, respectively, across cryptographic operations. Additionally, software evaluation as an ASIC shows further performance gains of up to 23.6x and 7.1x over FPGA implementation with 12nm and 28nm fabrication techniques, respectively.

2.6.2. SoC-based SmartNICs

Olteanu et al. [Olteanu et al. 2022] investigate challenges in data center networks, proposing the Edge-Queued Datagram Service (EQDS) to improve network utilization and support diverse transport protocols. EQDS moves queuing from switches to network edges, providing a datagram service via dynamic tunnels. Its receiver-driven control loop manages inbound traffic, ensuring isolation between protocols and facilitating fair sharing. EQDS offers improved protocol performance, protection from queuing delays, and increased throughput through load balancing. The study details EQDS design, implementation, and evaluation on Linux hosts and BlueField-2 SmartNIC, demonstrating its compatibility with existing transport protocols and its ability to enhance network performance.

Wei et al. [Wei et al. 2023c] delved into the performance characteristics of various communication paths on off-path SmartNICs, shedding light on essential aspects often overlooked in prior research. It highlighted the increasing adoption of RDMA in modern data centers, driving network bandwidth towards 400 Gbps. However, the intensified network speed demands more CPU resources to leverage RDMA-capable NICs, significantly burdening distributed systems fully. While one-sided RDMA can mitigate CPU pressures by allowing direct host memory access, limited offloading capabilities may lead to network amplification and performance degradation. Amidst these challenges, the emergence of SmartNICs with programmable capabilities offers a promising avenue for offloading complex computations. Two main types of SmartNICs are distinguished: on-path SmartNICs, which expose NIC cores for direct processing, and off-path SmartNICs, which employ programmable multicore SoCs adjacent to the RNIC cores. Due to their generality and programmability, the study primarily focused on off-path SmartNICs, such as NVIDIA Bluefield-2 and Broadcom Stingray. The study found that the RDMA path from the NIC to the SoC can be up to 1.48 times faster than the path to the host. It also revealed that RDMA requests involving the SoC may suffer from up to 48% bandwidth degradation due to performance anomalies introduced by the SoC.

Today, computing environments require seamless integration of hybrid environments spanning edge, cloud, and HPC systems, connecting sensors, elastic resources, and cloud-native frameworks with high-performance systems for efficient workflows. However, prevalent container networking architectures, reliant on overlay networks, often incur performance overhead, particularly affecting co-processes on the same node. To address this challenge, [Njavro et al. [Njavro et al. 2022] explore leveraging the Nvidia Bluefield 2 SmartNIC to offload Docker overlay networks, aiming to enhance support for cloud-native overlay networks alongside existing HPC workloads sensitive to system noise. Through characterizing different offloading approaches, investigating the feasibility of DPU offload, and evaluating performance benefits, the paper contributes to optimizing the integration of cloud-native technologies with traditional HPC environments, enabling efficient coexistence and operation of diverse workloads in hybrid computing infrastructures.

2.6.3. ASIC-based SmartNICs

Deploying machine learning algorithms in high-throughput networking environments poses significant challenges, with existing approaches often focusing on offline post-processing or introducing high latency. To address this, Xavier et al. [Xavier et al. 2021] introduce a framework to create simple yet accurate machine-learning models that are deployable directly into the data plane with acceptable performance degradation. Their approach involves translating these models, tailored for individual packets or flows, into the P4 language, a crucial step towards in-network deployment. Through validation using an intrusion detection use case and deployment on a Netronome SmartNIC (Agilio CX 2x10GbE), they demonstrate the feasibility of achieving high accuracy (above 95%) with minimal performance impact, even with many flows. This work signifies a step forward in integrating machine learning into network devices, leveraging the capabilities of programmable switches and SmartNICs to enhance real-world networking applications.

Integrating ML models into programmable networking devices has sparked interest in leveraging data plane capabilities for autonomous network management. However, challenges persist in adapting unsupervised ML algorithms to the constraints of programmable forwarding devices. In response, Cannarozzo et al [Cannarozzo et al. 2024] proposes SPINNER, an innovative approach for in-network flow clustering directly within the data plane. SPINNER maps network flows to multidimensional vectors and dynamically assigns them to clusters as packets traverse the programmable device. By clustering flows in the data plane, SPINNER enables line-rate flow balancing, congestion control, and differentiated services. Implemented in the SmartNIC Netronome NFP 4000, SPINNER demonstrates promising results, enhancing TCP throughput by up to 2X compared to vanilla TCP with minimal incurred latency.

The TCP protocol is the backbone of modern data networking, ensuring reliable data transfer between endpoints. Yet, its adherence to protocol standards often incurs performance penalties, particularly in short-lived connections and layer-7 proxying scenarios. Addressing this challenge, Moon et al. [Moon et al. 2020] leverages Netronome Agilio LX 40GbE to accelerate TCP processing, presenting a dual-stack design that offloads select operations to the NIC stack while maintaining control-plane functions at the host stack. This approach significantly reduces CPU and memory bandwidth overhead on the host stack, allowing applications to focus on core functionality. Moreover, Moon et al. [Moon et al. 2020] per-flow offloading decision offers flexibility, ensuring optimal performance under varying conditions. Despite challenges in maintaining consistency across host and NIC stacks, the work effectively manages complexity, achieving notable performance gains. Evaluation results demonstrate its superiority over existing TCP stacks and its substantial impact on real-world applications like Redis and HAProxy, making it a significant contribution to network performance optimization.

2.7. Diving into SmartNICs: Hands-on experiences

In this section, we are going to deep dive into practical hands-on examples using current available SmartNICs. Note that all used coding and additional scripting materials are available at our public repository.[10]

[10] `https://github.com/smartness2030/sbrc24-minicurso-smartnic`

2.7.1. Hands-on with Nvidia BlueField

In this example, we will demonstrate the step-by-step process of setting up a DOCA application. Firstly, we need to set up our environment correctly. Afterward, we will describe a simple DOCA Flow application that utilizes DOCA pipes to offload packet processing to the BlueField SmartNIC. It is important to note that our settings can also be replicated in available testbeds such as FABRIC.[11].

2.7.1.1. Setting up the SmartNIC

First, we need to install the DOCA framework. This is done on the host side using Ubuntu 20.04 (Kernel 5.15-0-67-generic). To install DOCA utils, download the file and follow the installation instructions.

```
smartness@host# dpkg -i doca-host-repo-ubuntu2204*_amd64.deb
smartness@host# apt-get update
smartness@host# apt install doca-runtime
smartness@host# apt install doca-tools
smartness@host# apt install -y doca-extra
smartness@host# apt install pv
```

Then, we need to initialize the Mellanox Software Tools service (a.k.a. MST). The MST command is used to create special files that represent Mellanox devices in the directory /dev/mst. This command loads appropriate kernel modules and saves PCI configuration headers in the temporary directory. Once this command is completed successfully, the MST driver is ready to work, and other Mellanox tools can be invoked.

```
smartness@host# mst start
smartness@host# mst status -v
```

The following is an example of the output you should expect to see when running the command `mst status -v`. The output will contain the PCI address (e.g. 03:00.0), the MST address (e.g. `mt41686_pciconf0`), and the name of the network interface that is exposed to the operating system (e.g. `net-enp3s0f0`).

```
MST modules:
------------
    MST PCI module is not loaded
    MST PCI configuration module loaded
PCI devices:
------------
DEVICE_TYPE          MST                           PCI       RDMA     NET
BlueField2(rev:1)    /dev/mst/mt41686_pciconf0.1   03:00.1   mlx5_1   net-enp3s0f1

BlueField2(rev:1)    /dev/mst/mt41686_pciconf0     03:00.0   mlx5_0   net-enp3s0f0
```

[11] `https://fabricmc.net/`

To get started, we must first initialize the RSHIM service. The RShim serves as the current interface for managing the SoC (System on a Chip). It enables an external agent, such as the host CPU or BMC, to operate the DPU (Data Processing Unit) and monitor its operational state. With this interface, the DPU can be provisioned, Arm cores can be reset, and logs can be obtained. On the host, the SoC management interface driver exposes a virtual Ethernet device called `tmfifo_net0`. This virtual Ethernet device functions as a peer-to-peer tunnel that links the host and the DPU OS. In response, the DPU OS sets up a comparable device. The BFB (Bootloader Firmware Binary) images within the DPU OS are tailored to set up the DPU end of this connection, assigning a predefined IP address of `192.168.100.2/30` within the DPU OS.

```
smartness@host# systemctl enable rshim
smartness@host# systemctl start rshim
smartness@host# systemctl status rshim
```

The following is the expected output of the RSHIM initialization mentioned above:

```
rshim.service - rshim driver for BlueField SoC
  Loaded: loaded (/lib/systemd/system/rshim.service; enabled; vendor preset: enabled)
  Active: active (running) since Wed 2024-04-03 13:19:25 -03; 4 days ago
          Docs: man:rshim(8)
  Main PID: 1322 (rshim)
      Tasks: 6 (limit: 18833)
      Memory: 1.1M
      CGroup: /system.slice/rshim.service
                 1322 /usr/sbin/rshim

abr 03 13:19:25 Hydra-202 systemd[1]: Starting rshim driver for BlueField SoC...
abr 03 13:19:25 Hydra-202 systemd[1]: Started rshim driver for BlueField SoC.
abr 03 13:19:25 Hydra-202 rshim[1322]: Probing pcie-0000:03:00.2
abr 03 13:19:25 Hydra-202 rshim[1322]: create rshim pcie-0000:03:00.2
abr 03 13:19:26 Hydra-202 rshim[1322]: rshim0 attached
```

If you are setting up the SmartNIC for the first time, you may need to install or reinstall the operating system in the SoC SmartNIC. To get started, you will need to download the Linux image and then use NVIDIA toolchain to install it. In this tutorial, we will be using Ubuntu `22.04`, which is provided by NVIDIA.

```
smartness@host# wget https://content.mellanox.com/BlueField/
                     BFBs/Ubuntu22.04/DOCA_2.0.2_BSP_4.0.3_
                     Ubuntu_22.04-10.23-04.prod.bfb
```

We need to set a configuration file and install the Ubuntu image with a predefined password using the `bfb-install` tool. The following steps will help you perform these actions.

```
smartness@host# hash=`openssl passwd -1`
smartness@host# echo ubuntu_PASSWORD='$hash' >> bf.cfg

smartness@host# bfb-install --rshim /dev/rshim0
                            --bfb <image_path.bfb>
                            --config bf.cfg
```

The operating system has been installed successfully and is now ready to be used. However, before using it, we should change the SmartNIC operation mode. For the purpose of this tutorial, we will be using the SmartNIC in the DPU mode. The NVIDIA BlueField offers three modes of operation: DPU mode (or embedded function ECPF), zero-trust mode, and NIC mode. In DPU mode, the NIC resources and functionalities are owned by the embedded Arm subsystem. All network communication destined for the host passes through a virtual switch control plane hosted on the Arm cores, before reaching the host. In this operational mode, the DPU acts as the trusted entity, overseen by both data center and host administrators. Its responsibilities include loading network drivers, resetting interfaces, toggling interface states, firmware updates, and altering the operational mode of the DPU device. The following steps will help you change the DPU operation mode.

```
smartness@host# #mst status -v
#Use the above command to check the MST device addrress

smartness@host# mlxconfig -d /dev/mst/mt41686_pciconf0
                s INTERNAL_CPU_MODEL=1
smartness@host# mlxconfig -d /dev/mst/mt41686_pciconf0.1
                s INTERNAL_CPU_MODEL=1
smartness@host# reboot
```

In order to access the SoC subsystem, we must configure a valid IP address for the virtual interface called `tmfifo_net0`. The SoC management interface has a peer-to-peer tunnel that connects the host and the DPU OS, and it is automatically set up in the `192.168.100.0/30` subnet. For this tutorial, use the following command to assign `192.168.100.2` to the `tmfifo_net0` interface.

```
smartness@host# ip address add 192.168.100.2/30 dev tmfifo_net0
```

Also, it is necessary to configure NAT rules on the host side to enable the OS in NIC to access the Internet. The following steps will configure `iptables` on the host side.

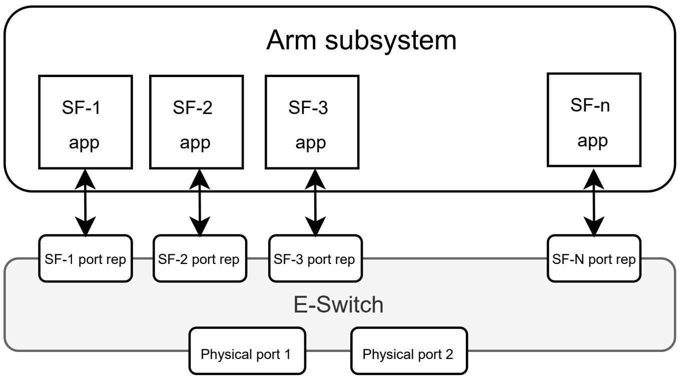


Figure 2.14. Scalable Function in the Nvidia BlueField SoC. Adapted from NVIDIA DOCA documentation [NVIDIA 2024].

```
#replace with the interface connected to the Internet
smartness@host# OUTFACE=enp6s0
smartness@host# NIC=tmfifo_net0
smartness@host# echo 1 > /proc/sys/net/ipv4/ip_forward
smartness@host# iptables -A FORWARD -i $NIC -o $OUTFACE \
                        -j ACCEPT
smartness@host# iptables -A FORWARD -i $OUTFACE -o $NIC \
                        -m state --state ESTABLISHED,RELATED \
                        -j ACCEPT
smartness@host# iptables -t nat -A POSTROUTING -o $OUTFACE \
                        -j MASQUERADE
```

2.7.1.2. Accessing the SoC and Preliminary Configurations

After configuring the host side, we can now access the SoC subsystem. To access the SmartNIC SoC operating system, we can use `ssh` as follows.

```
smartness@host# ssh ubuntu@192.168.100.2
```

If everything goes smoothly, you will be logged into the SmartNIC operating system. By default, the SmartNIC OS comes with an Open vSwitch (OVS) that has two default bridges named `ovsbr1` and `ovsbr2`. In each bridge, there is a physical port attached, namely `p1` and `p2`. Each OVS bridge has at least one host interface representator, such as `pf1hpf` and `pf0hpf`. These interface representators are used to connect with the host interfaces (e.g., `net-enp3s0f1`). Lastly, each OVS bridge comes with a Scalable Function (e.g., `en3f1pf1sf0` and `en3f0pf0sf0`). Scalable Functions (SFs), or sub-functions, are similar to Virtual Functions (VFs) that are part of a Single Root I/O Virtualization (SR-IOV) solution. An SF is a lightweight function that has a parent PCIe function on which it is deployed. The SF, therefore, has access to the capabilities and resources of its parent PCIe function, and it has its own function capabilities and resources. This means that an SF also has its own dedicated queues (i.e., txq, rxq). Figure 2.14 illustrates the relationship between internal ports and SF representators.

```
root@localhost:/home/ubuntu# ovs-vsctl show
e350e1eb-a1f2-4eb8-b9ad-8bb2a3a7f86a
    Bridge ovsbr2
        Port p1
            Interface p1
        Port pf1hpf
            Interface pf1hpf
        Port en3f1pf1sf0
            Interface en3f1pf1sf0
        Port ovsbr2
            Interface ovsbr2
                type: internal
    Bridge ovsbr1
        Port pf0hpf
            Interface pf0hpf
        Port ovsbr1
            Interface ovsbr1
                type: internal
        Port p0
            Interface p0
        Port en3f0pf0sf0
            Interface en3f0pf0sf0
    ovs_version: "2.17.7-e054917"
```

In Figure 2.14, you can see how physical ports interact with SFs. SFs are used by DOCA applications and communicate with the physical port by SF representators. It is possible to dynamically add, remove, or modify SFs according to the application's needs using the `mlxdevm port` command. Next, we can use the `mlxdevm port show` command to list all registered SFs and their features. For instance, we can see the SF identifier (e.g., 229408) and to which physical port each SF is attached. For example, the SF `pci/0000:03:00.0/229408` is attached to physical port 0.

```
root@smartNIC:/# /opt/mellanox/iproute2/sbin/mlxdevm port show
```

```
pci/0000:03:00.0/229408: type eth netdev en3f0pf0sf0 flavour pcisf controller 0
                                                             pfnum 0 sfnum 0
 function:
 hw_addr 02:eb:3d:64:53:86 state active opstate attached roce true max_uc_macs 128
                                                             trust off

pci/0000:03:00.1/294944: type eth netdev en3f1pf1sf0 flavour pcisf controller 0
                                                             pfnum 1 sfnum 0
 function:
 hw_addr 02:43:b8:2f:7a:fb state active opstate attached roce true max_uc_macs 128
                                                             trust off
```

Next, we will create, configure and deploy two Scalable Functions. For more information, please refer to DOCA Scalable Function documentation[12].

[12]`https://docs.nvidia.com/doca/archive/doca-v2.2.0/scalable-functions/index.html`

```
root@smartNIC:/# #Command syntax
                 #/opt/mellanox/iproute2/sbin/mlxdevm port \
                 function set pci/<pci_address>/<sf_index>  \
                 hw_addr <MAC address> trust on state active

root@smartNIC:/# /opt/mellanox/iproute2/sbin/mlxdevm port \
                 function set pci/0000:03:00.0/229409  \
                 hw_addr 00:00:00:00:04:0 trust on state active

root@smartNIC:/# /opt/mellanox/iproute2/sbin/mlxdevm port \
                 function set pci/0000:03:00.0/294944  \
                 hw_addr 00:00:00:00:05:0 trust on state active
```

We then verify if the just created SFs are correctly deployed using `devlink dev show` command.

```
root@smartNIC:/# devlink dev show
```

Last, we need to connect the SFs' interfaces to the OVS bridges. This will enable communication between SFs, physical interfaces, and the host interface representators.

```
root@smartNIC:/# ovs-vsctl add-port ovsbr1 en3f1pf1sf0
root@smartNIC:/# ovs-vsctl add-port ovsbr2 en3f1pf1sf1
root@smartNIC:/# ovs-vsctl show
```

In addition to that, we need to add flow entries to OVS (or to the eSwitch in the BlueField) so that packets can move between the physical interface and SFs.

```
root@smartNIC:/# sudo ovs-ofctl add-flow ovsbr1 in_port=p0,\
                 actions=output:en3f0pf0sf1
root@smartNIC:/# sudo ovs-ofctl add-flow ovsbr2 \
                 in_port=en3f1pf1sf1,actions=output:p1
root@smartNIC:/# sudo ovs-ofctl add-flow ovsbr2 in_port=p1,\
                 actions=output:en3f1pf1sf1
root@smartNIC:/# sudo ovs-ofctl add-flow ovsbr1 \
                 in_port=en3f0pf0sf1,actions=output:p0
root@smartNIC:/# sudo ovs-ofctl dump-flows ovsbr1
root@smartNIC:/# sudo ovs-ofctl dump-flows ovsbr2
```

2.7.1.3. Programming the SmartNIC with DOCA

We will now go over a simple DOCA application that is based on the DOCA Flow library. Our environment is now set up for the DOCA application. Our objective is to gain a high-level understanding of the code so that we can deploy and debug it with ease. As mentioned in Section 2.4, a DOCA Flow application is made up of pipes, where each pipe can have matching and actions. We will start by discussing the main building block of the sample code. You can find the complete code at our public repository[13].

[13] https://github.com/smartness2030/sbrc24-minicurso-smartnic

The first building block consists of a function that creates a DOCA Flow pipe. The function creates a simple pipe that matches on IPv4 (src_ip, dst_ip, src_port, dst_port). The function create_hairpin_pipe creates a hairpin pipe, which allows traffic to ingress and egress through the same physical port. It initializes various data structures such as struct doca_flow_match, struct doca_flow_actions, and struct doca_flow_fwd, and configures the pipe settings in struct doca_flow_pipe_cfg. Matching criteria are established for IPv4/TCP traffic with wildcard IP addresses and port numbers. Forwarding rules are set to direct traffic to another port. Finally, the function calls doca_flow_pipe_create to create the hairpin pipe based on the configured settings and returns the result.

```
static doca_error_t
create_hairpin_pipe(struct doca_flow_port *port, int port_id,
                    struct doca_flow_pipe **pipe)
{
        struct doca_flow_match match;
        struct doca_flow_actions actions, *actions_arr[NB_ACTIONS_ARR];
        struct doca_flow_fwd fwd;
        struct doca_flow_pipe_cfg pipe_cfg;

        memset(&match, 0, sizeof(match));
        memset(&actions, 0, sizeof(actions));
        memset(&fwd, 0, sizeof(fwd));
        memset(&pipe_cfg, 0, sizeof(pipe_cfg));

        pipe_cfg.attr.name = "HAIRPIN_PIPE";
        pipe_cfg.attr.type = DOCA_FLOW_PIPE_BASIC;
        pipe_cfg.match = &match;
        actions_arr[0] = &actions;
        pipe_cfg.actions = actions_arr;
        pipe_cfg.attr.is_root = true;
        pipe_cfg.attr.nb_actions = NB_ACTIONS_ARR;
        pipe_cfg.port = port;

        /* 5 tuple match */
        match.outer.l4_type_ext = DOCA_FLOW_L4_TYPE_EXT_TCP;
        match.outer.l3_type = DOCA_FLOW_L3_TYPE_IP4;
        match.outer.ip4.src_ip = 0xffffffff;
        match.outer.ip4.dst_ip = 0xffffffff;
        match.outer.tcp.l4_port.src_port = 0xffff;
        match.outer.tcp.l4_port.dst_port = 0xffff;

        /* forwarding traffic to other port */
        fwd.type = DOCA_FLOW_FWD_PORT;
        fwd.port_id = port_id ^ 1;

        return doca_flow_pipe_create(&pipe_cfg, &fwd, NULL, pipe);
}
```

Another important building block of of sample application is the function that adds pipe entries. Similarly to P4 (or OpenFlow), the function adds entries that are going to be used to match against the information of incoming packets. In the following example, we add a single entry having dst_ip = 8.8.8.8, src_ip = 1.2.3.4, dst_port = 80, and src_port = 1234. The function add_hairpin_pipe_entry is used to add an entry to a hairpin pipe in a networking context. It takes a struct doca_flow_pipe representing the pipe and a struct doca_flow_port representing the port as parameters. Inside the function, it initializes several data structures including struct doca_flow_match and struct doca_flow_actions. It sets up matching criteria for IPv4/TCP traffic with specific source and destination IP addresses and port numbers. The function then adds this entry to the pipe using doca_flow_pipe_add_entry. Subsequently, it processes the added entry on the specified port using doca_flow_entries_process and checks the processing status. This function processes entries in the queue. The application must invoke this function to complete flow rule offloading and to receive the flow rule's operation status. If the entry is successfully processed, it returns DOCA_SUCCESS, otherwise, it returns an appropriate error code.

```
static doca_error_t
add_hairpin_pipe_entry(struct doca_flow_pipe *pipe, struct doca_flow_port *port)
{
        struct doca_flow_match match;
        struct doca_flow_actions actions;
        struct doca_flow_pipe_entry *entry;
        struct entries_status status;
        doca_error_t result;
        int num_of_entries = 1;

        /* example 5-tuple to forward */
        doca_be32_t dst_ip_addr = BE_IPV4_ADDR(8, 8, 8, 8);
        doca_be32_t src_ip_addr = BE_IPV4_ADDR(1, 2, 3, 4);
        doca_be16_t dst_port = rte_cpu_to_be_16(80);
        doca_be16_t src_port = rte_cpu_to_be_16(1234);

        memset(&status, 0, sizeof(status));
        memset(&match, 0, sizeof(match));
        memset(&actions, 0, sizeof(actions));

        match.outer.ip4.dst_ip = dst_ip_addr;
        match.outer.ip4.src_ip = src_ip_addr;
        match.outer.tcp.l4_port.dst_port = dst_port;
        match.outer.tcp.l4_port.src_port = src_port;

        result = doca_flow_pipe_add_entry(0, pipe, &match, &actions, NULL, NULL,
                0, &status, &entry);
        if (result != DOCA_SUCCESS)
                return result;

        result = doca_flow_entries_process(port, 0, DEFAULT_TIMEOUT_US
                                           , num_of_entries);
        if (result != DOCA_SUCCESS)
                return result;

        if (status.nb_processed != num_of_entries || status.failure)
                return DOCA_ERROR_BAD_STATE;

        return DOCA_SUCCESS;
}
```

Then, we initialize the DOCA Flow API function and call the previously defined functions to create and populate DOCA pipes. In this example, we create a pipe for each existing port. The flow_hairpin function orchestrates the configuration of hairpin networking scenarios using the DOCA library. It initializes the necessary DOCA flow resources and ports, then enters a loop where it iterates through each port, creating a hairpin pipe and adding an entry to it. After each iteration, it waits for a brief period for packets to arrive. This process continues indefinitely until manually stopped. Upon completion, it properly stops the DOCA flow ports and releases the associated resources.

```
#include <string.h>
#include <unistd.h>
#include <rte_byteorder.h>
#include <doca_log.h>
#include <doca_flow.h>
#include "flow_common.h"

DOCA_LOG_REGISTER(FLOW_HAIRPIN);

doca_error_t
flow_hairpin(int nb_queues)
{
        int inc = 0;
        int nb_ports = 2;
        struct doca_flow_resources resource = {0};
        uint32_t nr_shared_resources[DOCA_FLOW_SHARED_RESOURCE_MAX] = {0};
        struct doca_flow_port *ports[nb_ports];
        struct doca_flow_pipe *pipe;
        doca_error_t result;
        int port_id;

        result = init_doca_flow(nb_queues, "vnf,hws", resource, nr_shared_resources);
        if (result != DOCA_SUCCESS) {
                DOCA_LOG_ERR("Failed to init DOCA Flow: %s",
                             doca_get_error_string(result));
                return result;
        }

        result = init_doca_flow_ports(nb_ports, ports, true);
        if (result != DOCA_SUCCESS) {
                DOCA_LOG_ERR("Failed to init DOCA ports: %s",
                             doca_get_error_string(result));
                doca_flow_destroy();
                return result;
        }

        while(1){
        for (port_id = 0; port_id < nb_ports; port_id++) {
                result = create_hairpin_pipe(ports[port_id], port_id, &pipe);
                if (result != DOCA_SUCCESS) {
                        DOCA_LOG_ERR("Failed to create pipe: %s",
                                     doca_get_error_string(result));
                        stop_doca_flow_ports(nb_ports, ports);
                        doca_flow_destroy();
                        return result;
                }

                result = add_hairpin_pipe_entry(pipe, ports[port_id]);
                if (result != DOCA_SUCCESS) {
                        DOCA_LOG_ERR("Failed to add entry: %s",
                                      doca_get_error_string(result));
                        stop_doca_flow_ports(nb_ports, ports);
                        doca_flow_destroy();
                        return result;
                }
        }

        DOCA_LOG_INFO("tst2: %i", inc);
        DOCA_LOG_INFO("Wait few seconds for packets to arrive");
        sleep(5);
        }

        stop_doca_flow_ports(nb_ports, ports);
        doca_flow_destroy();
        return DOCA_SUCCESS;
}
```

Overall, the function provides a streamlined approach to setting up and managing hairpin configurations across multiple ports for networking tasks. The full code is available at our GitHub.[14] Next, we compile and run the code. To compile, we must ensure that DOCA is on the Linux path before running the code.

```
root@smartNIC:/# export PKG_CONFIG_PATH=:/opt/mellanox/doca/ \
                 lib/aarch64-linux gnu/pkgconfig:/opt/ \
                 mellanox/dpdk/lib/aarch64-linux-gnu/pkgconfig \
                 \:/opt/mellanox/flexio/lib/pkgconfig
```

Then, we compile with Meson and execute the DOCA application.

```
root@smartNIC:/app # meson build
root@smartNIC:/app # cd build
root@smartNIC:/app/build # ninja
root@smartNIC:/app/build # ./doca_flow_hairpin \
            -a auxiliary:mlx5_core.sf.4,dv_flow_en=2 \
            -a auxiliary:mlx5_core.sf.5,dv_flow_en=2 -- -l 60
```

Once we have done that, we can start sending packets to our DOCA application. Figure 2.15(a) illustrates the TRex Traffic Generator[15] sending TCP packets to the BlueField-2 Soc. Observe that the BlueField-2 is handling almost 30Mpps (millions of packet per second) in both directions. As the whole packet processing is offloaded using the DOCA application discussed, we observe that the ARM cores in the SoC subsystem are completely not used (see Figure 2.15(b).

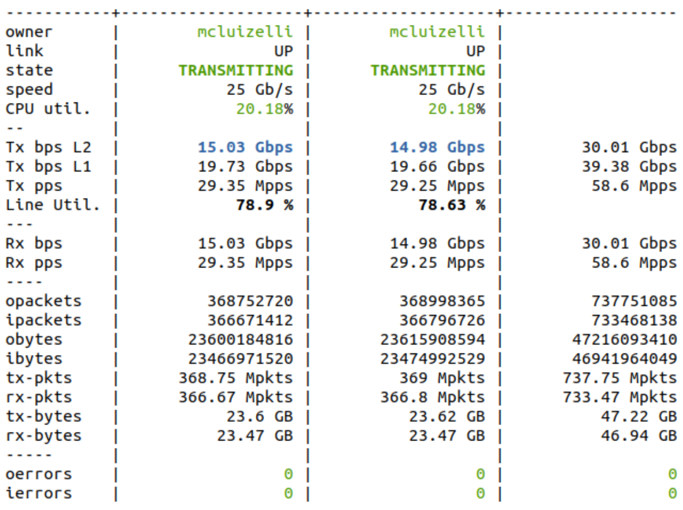
2.7.2. Hands-on with Nvidia Connectx SmartNIC offload capabilities

In this example, we will showcase the offload capabilities of the Nvidia Connectx SmartNIC. We will compare the performance of a software router with and without offloads. To implement the software router, we will use Vector Packet Processing (VPP) [Barach et al. 2018], a high-performance network stack that supports different data planes (Linux, RDMA, and DPDK). VPP can be used as vSwitches, vRouters, Gateways, Firewalls, and Load-Balancers. The reference traffic generator from DPDK, Pktgen-DPDK, will be used to stress test VPP. It can generate 100Gbps using a single CPU core. We will conduct the tests using Nvidia Connectx-6 SmartNIC, which is available on testbeds like FABRIC and RNP Testbed Service[16] to facilitate replication of this example.

2.7.2.1. Setting up the VPP SmartNIC

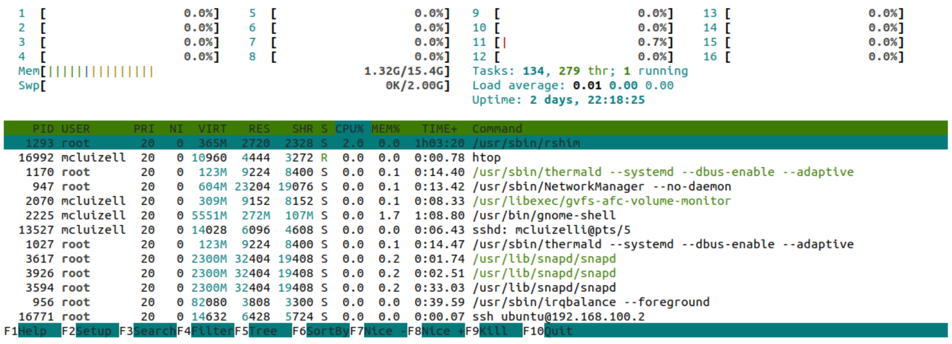
To begin with, you need to recognize the ConnectX SmartNIC network interfaces and take note of their name, PCI address, and MAC address. To list all network interfaces, including the necessary details, you can use the following show command. However, only rows 3 and 4 will be utilized, which correspond to ConnectX-6 physical interfaces.

[14]`https://github.com/smartness2030/sbrc24-minicurso-smartnic`
[15]`https://trex-tgn.cisco.com/`
[16]`https://www.rnp.br/en/research-development/testbeds`



```
-----------+-------------------+-------------------+------------------
owner      |        mcluizelli |        mcluizelli |
link       |                UP |                UP |
state      |      TRANSMITTING |      TRANSMITTING |
speed      |           25 Gb/s |           25 Gb/s |
CPU util.  |            20.18% |            20.18% |
--         |                   |                   |
Tx bps L2  |        15.03 Gbps |        14.98 Gbps |        30.01 Gbps
Tx bps L1  |        19.73 Gbps |        19.66 Gbps |        39.38 Gbps
Tx pps     |        29.35 Mpps |        29.25 Mpps |         58.6 Mpps
Line Util. |            78.9 % |           78.63 % |
---        |                   |                   |
Rx bps     |        15.03 Gbps |        14.98 Gbps |        30.01 Gbps
Rx pps     |        29.35 Mpps |        29.25 Mpps |         58.6 Mpps
----       |                   |                   |
opackets   |         368752720 |         368998365 |         737751085
ipackets   |         366671412 |         366796726 |         733468138
obytes     |       23600184816 |       23615908594 |       47216093410
ibytes     |       23466971520 |       23474992529 |       46941964049
tx-pkts    |       368.75 Mpkts |         369 Mpkts |       737.75 Mpkts
rx-pkts    |       366.67 Mpkts |       366.8 Mpkts |       733.47 Mpkts
tx-bytes   |           23.6 GB |          23.62 GB |          47.22 GB
rx-bytes   |          23.47 GB |          23.47 GB |          46.94 GB
-----      |                   |                   |
oerrors    |                 0 |                 0 |                 0
ierrors    |                 0 |                 0 |                 0
```

(a) TRex Traffic Generator when running sample application on BlueField-2.



```
  1  [                  0.0%]   5  [                    0.0%]   9  [                  0.0%]   13 [                    0.0%]
  2  [                  0.0%]   6  [                    0.0%]   10 [                  0.0%]   14 [                    0.0%]
  3  [                  0.0%]   7  [                    0.0%]   11 [|                 0.7%]   15 [                    0.0%]
  4  [                  0.0%]   8  [                    0.0%]   12 [                  0.0%]   16 [                    0.0%]
  Mem[|||||||||||||||                             1.32G/15.4G]   Tasks: 134, 279 thr; 1 running
  Swp[                                              0K/2.00G]   Load average: 0.01 0.00 0.00
                                                                Uptime: 2 days, 22:18:25

    PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
   1293 root       20   0  365M  2720  2328 S  2.0  0.0  1h03:20 /usr/sbin/rshim
  16992 mcluizell  20   0 10960  4444  3272 R  0.0  0.0  0:00.78 htop
   1170 root       20   0  123M  9224  8400 S  0.0  0.1  0:14.40 /usr/sbin/thermald --systemd --dbus-enable --adaptive
    947 root       20   0  604M 23204 19076 S  0.0  0.1  0:13.42 /usr/sbin/NetworkManager --no-daemon
   2070 mcluizell  20   0  309M  9152  8152 S  0.0  0.1  0:08.33 /usr/libexec/gvfs-afc-volume-monitor
   2225 mcluizell  20   0 5551M  272M  107M S  0.0  1.7  1:08.80 /usr/bin/gnome-shell
  13527 mcluizell  20   0 14028  6096  4608 S  0.0  0.0  0:06.43 sshd: mcluizelli@pts/5
   1027 root       20   0  123M  9224  8400 S  0.0  0.1  0:14.47 /usr/sbin/thermald --systemd --dbus-enable --adaptive
   3617 root       20   0 2300M 32404 19408 S  0.0  0.2  0:01.74 /usr/lib/snapd/snapd
   3926 root       20   0 2300M 32404 19408 S  0.0  0.2  0:02.51 /usr/lib/snapd/snapd
   3594 root       20   0 2300M 32404 19408 S  0.0  0.2  0:33.03 /usr/lib/snapd/snapd
    956 root       20   0 82080  3808  3300 S  0.0  0.0  0:39.59 /usr/sbin/irqbalance --foreground
  16771 root       20   0 14632  6428  5724 S  0.0  0.0  0:00.07 ssh ubuntu@192.168.100.2
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

(b) BlueField-2 SoC CPU usage using `htop` command in Linux.

Figure 2.15. Running sample application on BlueField-2 SoC.

```
ubuntu@vpp-node:~$ sudo lshw -class network -businfo
```

```
Bus info          Device      Class          Description
==================================================
pci@0000:03:00.0              network        Virtio network device
virtio@1          enp3s0      network        Ethernet interface
pci@0000:07:00.0  enp7s0np0   network        MT28908 Family [ConnectX-6]
pci@0000:08:00.0  enp8s0np0   network        MT28908 Family [ConnectX-6]
pci@0000:09:00.0  enp9s0      network        MT28908 Family
```

The first column of the output displays the PCI address, which is used for both `VPP_IF#_PCI` and `VPP_IF#_NAME`. `VPP_IF#_NAME` is used as the interface name on VPP when using the DPDK backend. The second column represents the interface name on Linux, which is denoted by `VPP_IF#`.

```
ubuntu@vpp-node:~$ VPP_IF1_PCI=07:00.0
ubuntu@vpp-node:~$ VPP_IF1_NAME="HundredGigabitEthernet7/0/0"
ubuntu@vpp-node:~$ VPP_IF1=enp7s0np0
ubuntu@vpp-node:~$ VPP_IF2=enp8s0np0
ubuntu@vpp-node:~$ VPP_IF2_NAME="HundredGigabitEthernet8/0/0"
ubuntu@vpp-node:~$ VPP_IF2_PCI=08:00.0
```

As an initial baseline, we will run VPP without any offload, using only the Linux networking stack. To increase performance, we need to configure the IP address on both interfaces and set the MTU to 9000.

```
ubuntu@vpp-node:~$ sudo ifconfig $VPP_IF1 mtu 9000
ubuntu@vpp-node:~$ sudo ifconfig $VPP_IF1 192.168.0.2/24
ubuntu@vpp-node:~$ sudo ifconfig $VPP_IF2 mtu 9000
ubuntu@vpp-node:~$ sudo ifconfig $VPP_IF2 192.168.1.2/24
```

Enabling IP Forwarding is necessary to route packets through different interfaces. If using a firewall on your Linux host, consider adding the following rules using `nft` or `iptables`.

```
ubuntu@vpp-node:~$ sudo sysctl -w net.ipv4.ip_forward=1
```

```
# nft rules
ubuntu@vpp-node:~$ sudo nft insert rule ip filter \
                   FORWARD iifname $VPP_IF1 oifname $VPP_IF2 \
                   counter accept
ubuntu@vpp-node:~$ sudo nft insert rule ip filter \
                   FORWARD iifname $VPP_IF2 oifname $VPP_IF1 \
                   counter accept

# iptables rules
ubuntu@vpp-node:~$ sudo iptables -A FORWARD -i $VPP_IF1 \
                   -o $VPP_IF2 -j ACCEPT
ubuntu@vpp-node:~$ sudo iptables -A FORWARD -i $VPP_IF2 \
                   -o $VPP_IF1 -j ACCEPT
```

For this example, we will be using Version 24.02 of VPP. To deploy and run it, you can use the docker compose file provided in the `vpp` folder of the GitHub repository. Initially, VPP will run with a custom configuration file called `startup-nodpdk.conf` which disables DPDK and enables the use of other datapaths.

```
...
plugins {
        ## Enable all plugins by default and then selectively disable specific plugins
        plugin dpdk_plugin.so { disable }
}
...
```

Start VPP container in the background using docker compose up with “-d” flag

```
ubuntu@vpp-node:~$ docker compose up -d
```

To simplify running `vppctl` from within the container, create an alias using the command line. This tool is used to configure VPP, including the interfaces and IP addresses, based on the variables set earlier in this section. If you want to use native Linux interfaces on VPP without any additional offload, you can specify the `host-interface` type.

```
ubuntu@vpp-node:~$ alias vppctl="docker compose exec vpp vppctl"
ubuntu@vpp-node:~$ vppctl create host-interface name $VPP_IF1
ubuntu@vpp-node:~$ vppctl set int ip address host-$VPP_IF1 \
                   192.168.0.2/24
ubuntu@vpp-node:~$ vppctl set interface state host-$VPP_IF1 up
ubuntu@vpp-node:~$ vppctl create host-interface name $VPP_IF2
ubuntu@vpp-node:~$ vppctl set int ip address host-$VPP_IF2 \
                   192.168.1.2/24
ubuntu@vpp-node:~$ vppctl set interface state host-$VPP_IF2 up
```

Use the following command to confirm that the interface name, IP address, and state are as intended.

```
ubuntu@vpp-node:~$ vppctl show interface address
```

```
host-enp7s0np0 (up):
  L3 192.168.0.2/24
host-enp8s0np0 (up):
  L3 192.168.1.2/24
local0 (dn):
```

In the following section, `VPP_IF1 MAC` address needs to be configured on Pktgen-DPDK, so take note of it using the `ip link` command and referring to the value just after `link/ether`, which in the following example is `10:70:fd:e5:cd:60`.

```
ubuntu@vpp-node:~$ ip link show $VPP_IF1
```

```
3: enp7s0np0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9000 qdisc mq state UP mode \
              DEFAULT group default qlen 1000
    link/ether 10:70:fd:e5:cd:60 brd ff:ff:ff:ff:ff:ff
```

2.7.2.2. Setting up the Pktgen-DPDK SmartNIC

DPDK applications use large blocks of contiguous memory called `hugepages`. These blocks can be either 2MB or 1GB in size. For instance, to run Pktgen-DPDK, we need to allocate 2048 blocks of 2MB in size.

```
ubuntu@pktgen-node:~$ echo 2048 | sudo tee /sys/kernel/mm/hugepages/ \
                      hugepages-2048kB/nr_hugepages
```

Next, we have to create a container that has Pktgen-DPDK 24.03.1 and DPDK 24.03 by utilizing the Dockerfile in the `pktgen-dpdk` folder of the GitHub repository. We can leverage Docker Compose to build the Pktgen-DPDK container.

```
ubuntu@pktgen-node:~$ docker compose build
```

Pktgen-DPDK is an interactive application. To enable it to refresh the screen and display real-time information, we use the `docker-compose run` command. We also need to provide Pktgen-DPDK with the PCI address of the interfaces and the CPU cores that we intend to use.

```
ubuntu@pktgen-node:~$ docker compose run --rm pktgen-dpdk \
                      pktgen -a 08:00.0 -a 09:00.0 -l 0-4 \
                      -n 3 -- -m "[1:3].0, [2:4].1" -j
```

Afterward, Pktgen-DPDK will provide a prompt where we can set the parameters of the traffic we want to generate. In this example, we will set the protocol to UDP, the packet size to 9000, and the destination MAC address to `VPP_IF1` MAC address noted at the end of section 2.8.2.1. The command `start 0` initiates the packet generation on port 0 using the provided profile. Additionally, `enable 1 process` will allow port 1 to reply to ARP requests, which are necessary to establish end-to-end communication.

```
Pktgen:/> set 0 proto udp
Pktgen:/> set 0 size 9000
Pktgen:/> set 0 dst mac 10:70:fd:e5:cd:60
Pktgen:/> start 0
Pktgen:/> enable 1 process
```

Pktgen-DPDK will start updating the statistics on the top of the page, where we can track the transmitted rate (TX) on port 0 and the receive rate on port 1 (RX) on line 6 (MBits/s Rx/Tx). In the following example, Pktgen-DPDK is transmitting 98.4 Gbps (100753 Mbps / 1024) and receiving only 17 Gbps (17468 Mbps / 1024) using the Linux datapath.

```
Ports 0-1 of 2   <Main Page>  Copyright(c) <2010-2023>, Intel Corporation
  Port:Flags        : 0:P------      Single 1:P--I---      Single
Link State          :       <UP-100000-FD>        <UP-100000-FD>
---Total Rate---
Pkts/s Rx           :                    0              242191
242191
       Tx           :              1396864                   0
1396864
MBits/s Rx/Tx       :             0/100753             17468/0
17468/100753
...
```

It is important to keep Pktgen running because we will use it to verify the throughput of the SmartNIC using offloads with RDMA and DPDK.

2.7.2.3. Setting up the VPP SmartNIC with RDMA offload

VPP provides a driver for RDMA (Remote Direct Memory Access) devices[17], which uses RDMA APIs (Application Programming Interfaces) that are available on Nvidia ConnectX SmartNICS. This driver helps to offload Ethernet packets. To use it, you need to restart the VPP container to clean up the configurations and configure the RDMA interfaces.

```
docker compose restart
```

In this example, RDMA type interfaces are created to use the RDMA device driver. Physical interfaces are referenced using the `VPP_IF#` variable defined earlier. You can use the following command to confirm that the interface name, IP address, and state are as intended.

```
ubuntu@vpp-node:~$ alias vppctl="docker compose exec vpp vppctl"
ubuntu@vpp-node:~$ vppctl create interface rdma host-if $VPP_IF1 name rdma-0
ubuntu@vpp-node:~$ vppctl set int ip address rdma-0 192.168.0.2/24
ubuntu@vpp-node:~$ vppctl set interface state rdma-0 up
ubuntu@vpp-node:~$ vppctl create interface rdma host-if $VPP_IF2 name rdma-1
ubuntu@vpp-node:~$ vppctl set int ip address rdma-1 192.168.1.2/24
ubuntu@vpp-node:~$ vppctl set interface state rdma-1 up
```

Again, use the following command to confirm that the interface name, IP address, and state are as intended.

```
ubuntu@vpp-node:~$ vppctl show interface address
```

[17] `https://fd.io/docs/vpp/master/developer/devicedrivers/rdma.html`

```
local0 (dn):
rdma-0 (up):
  L3 192.168.0.2/24
rdma-1 (up):
  L3 192.168.1.2/24
```

Once you have confirmed that everything is correctly set up, go back to Pktgen-DPDK and verify that the receiving rate has increased by a factor of approximately 2.6 times to 44 Gbps (45348 Mbps / 1024).

```
| Ports 0-1 of 2   <Main Page>  Copyright(c) <2010-2023>, Intel Corporation
  Port:Flags          : 0:P------       Single 1:P--I---      Single
Link State            :         <UP-100000-FD>        <UP-100000-FD>
---Total Rate---
Pkts/s Rx             :                      0               628729
628729
       Tx             :                1385728                    0
1385728
MBits/s Rx/Tx         :                0/99949              45348/0
45348/99949
```

2.7.2.4. Setting up the VPP SmartNIC with DPDK offload

In this section, we will be using DPDK to accelerate VPP. As previously mentioned, we need to allocate hugepages to use DPDK.

```
ubuntu@vpp-node:~$ echo 2048 |
                   sudo tee /proc/sys/vm/nr_hugepages
```

To begin with, stop the VPP container and run it by explicitly setting the `compose-yaml` file using “-f”. This will enable VPP’s default configuration file and initialize it with the DPDK plugin.

```
ubuntu@vpp-node:~$ docker compose down
ubuntu@vpp-node:~$ docker compose -f compose.yaml up -d
```

VPP will automatically detect and create interfaces that are compatible with the DPDK plugin. You can verify this by using the following command.

```
ubuntu@vpp-node:~$ alias vppctl="docker compose exec vpp vppctl"
ubuntu@vpp-node:~$ vppctl show interface
```

```
              Name                    Idx     State           MTU
HundredGigabitEthernet7/0/0            1      down         9000/0/0/0
HundredGigabitEthernet8/0/0            2      down         9000/0/0/0
HundredGigabitEthernet9/0/0/4096       3      down         9000/0/0/0
local0                                 0      down          0/0/0/0
```

Next, configure the IP address and interface state by using the `VPP_IF#_NAME` variable that was set before.

```
ubuntu@vpp-node:~$ alias vppctl="docker compose exec vpp vppctl"
ubuntu@vpp-node:~$ vppctl set int ip address $VPP_IF1_NAME 192.168.0.2/24
ubuntu@vpp-node:~$ vppctl set interface state $VPP_IF1_NAME up
ubuntu@vpp-node:~$ vppctl set int ip address $VPP_IF2_NAME 192.168.1.2/24
ubuntu@vpp-node:~$ vppctl set interface state $VPP_IF2_NAME up
```

Finally, return to Pktgen-DPDK and check if the receiving rate has increased by approximately 5.8 times compared to the original test, to 99.2 Gbps (101543 Mbps / 1024).

```
0-1 of 2   <Main Page>  Copyright(c) <2010-2023>, Intel Corporation
  Port:Flags         : 0:P------       Single  1:P--I---      Single
Link State           :        <UP-100000-FD>        <UP-100000-FD>
---Total Rate---
Pkts/s Rx            :                    0              1407830
1407830
       Tx            :              1408000                    0
1408000
MBits/s Rx/Tx        :             0/101556             101543/0
101543/101556
...
```

2.7.3. Hands-on with Netronome Agilio CX

This tutorial provides a basic introduction to P4 programming on Netronome SmartNICs Agilio CX 2x10Gb. Here, you will find all the steps for installing and setting up the development environment, as well as presenting, implementing, and running a simple P4 program on a Netronome SmartNIC. The goal is to provide a quick learning curve by covering only the essential topics up to the implementation of the first program. To get complete step-by-step instructions, please refer to our public repository.

2.7.3.1. Installation and Environment Setup

To install the necessary drivers and modules and configure the environment, follow the instructions below, as root:

```
smartness@host# cd ~/
smartness@host# git clone https://github.com/ \
                guimvmatos/SBRC24NetronomeTutorial.git
```

After this, access the Agilio-P4-SmartNIC directory and follow the tutorial instructions to install the required drivers and modules. By following these steps, you will be ready to start developing and running your P4 programs on Netronome SmartNIC. In the installation directory, you will find a lot of Netronome related documents, that can be useful in case you want to deep into this technology.

2.7.3.2. Details about Netronome architecture

The Netronome SmartNIC uses single-root input/output virtualization (SRIOV), which enables virtual functions (VFs) to be created from a physical function (PF). The VFs thus share the resources of a PF, while VFs remain isolated from each other. The isolated VFs are typically assigned to virtual machines (VMs) on the host. This way, the VFs allow the VMs to directly access the PCI device, bypassing the host kernel. In this tutorial, we have two physical (`p0`, `p1`) and four virtual interfaces (`Vf0.0` to `Vf0.3`). We will work with a P4 program that implements simple IPv6 forwarding, which can be found at `IPv6Forwarding`.

2.7.3.3. Deployment

Here we will show how to deploy, configure, and debug your programs on Netronome SmartNICs. We are taking into consideration that you already clone this repository and that your environment is configured. Once you have your Netronome configured on your machine, you need to locate the `src/p4-16` folder inside the `Agilio-P4-SmartNIC` directory. To get started, ensure you are inside the `p4-16` folder, and then create a folder called `SimpleIPv6`. After that, copy the contents of the `IPv6Forwarding` folder into the folder you just created:

```
smartness@host# cd /root/SBRC24NetronomeTutorial/Agilio-P4-SmartNIC/src/p4-16
smartness@host# mkdir SimpleIPv6 \&\& cd SimpleIPv6/
smartness@host# cp ../../../../IPv6Forwarding/* ./
```

For P4 programming using Netronome, these files are all that is needed. The `ipv6_forwarding.p4` file represents our program itself. The `user_config.json` is the file that will populate the control plane tables. As this tutorial is for those who are already familiar with the P4 programming language, we will not go into details. However, a very common issue for those who are starting to program using Netronome SmartNICs is how to perform interface assignments in the control plane table. When we are working with Netronome SmartNICs, we have physical and virtual interfaces, and you can configure this setting by editing `/lib/systemd/system/nfp-sdk6-rte.service`. Locate and change the following line.

```
Environment=NUM_VFS=4
```

With this configuration, you will have 4 virtual interfaces called VFs. And they can be instantiated in the control plane tables as `v0.0`, `v0.1`, `v0.2`, and `v0.3`. If you want to use the physical interfaces, you should refer to them as `p0` and `p1`. In the `user_config.json` file, we can find examples of how to populate the control plane tables using the aforementioned nomenclature. Now that we have copied the two necessary files, we should call the compiler passing the P4 file as a parameter to create firmware. Then, we will deploy the firmware to the board, and finally, we will populate the control plane tables. To do this, execute the following commands:

```
smartness@host# sudo ./opt/netronome/p4/bin/nfp4build
                     --nfp4c\p4\_version 16
                     --no-debug-info -p out -o firmware.nffw
                     -l lithium -4 ipv6\_forward.p4

smartness@host# sudo ./opt/netronome/p4/bin/rtecli design-load
                    -f firmware.nffw -p out/pif\_design.json

smartness@host# sudo /opt/netronome/p4/bin/rtecli \
                     config-reload -c user\_config.json
```

If you encounter issues with initializing the service or deploying the program, you can find the logs in `/var/log/nfp-sdk6-rte.log` and better understand what is happening. If you want, you can check the configured rules about the control plane configuration.

```
smartness@host#sudo ./opt/netronome/p4/bin/rtecli tables
                    -i 0 list-rules
```

Now that your board is properly configured, to run the test programs, open more two shells, navigate to `/Agilio-P4-SmartNIC/src/p4-16/SimpleIPv6` directory, and in the first shell, run the command `python3 receive.py`. The program will execute and wait for any packet received on interface `v0.3`. After that, in the second terminal, run `python3 send_pkt.py`. A packet will be sent on interface v0.0 destined for v0.3. If the execution of the programs was successful, congratulations! You have completed this tutorial, and the program is ready for the next step. If you are interested in delving deeper and performing more advanced functions, please refer to our public repository. There, you will also find a tutorial for deploying a more advanced program, using SRv6 to connect traffic from multiple virtual machines.

2.8. Challenges and Future Trends

Standartization of SmartNICs. Despite existing efforts towards defining a common NIC architecture [(PNA) 2023] and programming framework such as P4 language, we are still far from having standard programming interfaces fully available in all SmartNICs. As we have seen in previous sections, there are multiple ways to program SmartNICs. The lack of proper programming standards across multiple SmartNIC vendors hinders the wide adoption of network offloaded solutions and suffers from interoperability hazards amongst hardware platforms.

Variable Performance. Packet processing performance in SmartNIC is subject to significant variation across programs, traffic types, and table entries – and, therefore, line-speed processing is not an automatic guarantee. There are two reasons for that. As discussed, SmartNICs usually follow a processing model where a packet is assigned to a particular processing engine in a run-to-completion manner. Second, existing P4 compilers focus on switch ASICs, where resource constraints are the first-order concern, and performance is guaranteed as long as the packed program fits inside the device. [Xing et al. 2023] argue that P4 compilers need to be revised in order to consider SmartNIC architecture nature. A few studies have been done to understand the performance of SmartNICs offloaded programs (e.g., [Viegas et al. 2021, Katsikas et al. 2021]). However, programmers still need to know hardware platform details to extract the most out of the offloaded program. In this context, automatic performance estimation of SmartNIC code would help programmers reduce the time developing/testing applications.

Limited Resource Orchestration. SmartNIC can run different offloaded codes from different applications/tenants. That can be achieved using SoC cores, or specialized-ASIC processing units. Despite a few related studies [Saquetti et al. 2020], little has yet been done to provide a unified resource orchestration layer for SmartNICs, allowing to run multiple isolated applications on top of the same platform. NVIDIA BlueField, for instance, allows running multiple DOCA applications in the SoC. However, there is little support to isolate them with current virtualization techniques. Others vendors, such as Netronome, do not provide any support for isolation, or resource orchestration. Therefore, as SmartNICs follow a run-to-completion model, an application can interfere with the performance of others running on the same card.

Limited Programmability. Each architecture provides a set of programmable primitives and constraints. For instance, Netronome provided limited coherent access to external memory and a lack of a complex arithmetic logic unit. These constraints lead to some workarounds, like employing fixed-point representations of real numbers. These methods can constrain the accuracy of the computing performed in the data plane. In turn, FPGA-based SmartNICs can provide rich hardware primitives as far as the hardware description can be synthesized. In addition to the aforementioned limited programmability, debugging/troubleshooting them also requires additional efforts from programmers.

Application offloading. SmartNICs and hardware accelerations enable a plethora of system optimizations. However, it is still not clear which application to offload and, more importantly, which parts of the application to offload. In part, this is due to the existing hardware limitations (e.g., not all applications benefit from running in a SmartNIC), and the diversity of existing hardware architectures and programming tools. Defining a methodology or a guideline is crucial to ensure an optimal matching between the application's needs and networking accelerators. Taking a packet processing application, it is hard to grasp which part (if any) is supposed to be offloaded, and more importantly to which acceleration technology: hardware SmartNIC, or DPDK, or XDP/eBPF, or a combination of them. For example, [Xing et al. 2023] proposes to use Profile-Guided Optimization to tailor P4 code to the underlying SmartNIC hardware enabling better performance optimizations by leveraging knowledge about the input. Despite this effort, we still need better development tools (e.g., specific compilers) to automatically support network technology developers.

Opportunities with Compute eXpress Link (CXL). The Compute Express Link (CXL) is an open industry-standard interconnect between processors and devices such as accelerators, memory buffers, smart network interfaces, persistent memory, and solid-state drives. Since its first release in 2019, CXL has evolved through three generations. CXL offers coherency and memory semantics with bandwidth that scales with PCIe capacity while achieving significantly lower latency than PCIe [Li et al. 2023]. In short, CXL focus on tackling the following challenges: (i) coherent access to system and device memory; (ii) memory scalability; (iii) inefficient usage of CPU/memory due to stranded resources; and (iv) fine-grained data sharing in distributed systems. Non-coherent accesses work well for streaming I/O operations such as storage access. In the context of networking accelerators (e.g., FPGAs or SoCs), entire data structures are moved from system memory to the accelerator for specific functions before being moved back to the main memory and software mechanisms are used to avoid simultaneous accesses between CPUs and accelerators. The usage of CXL will enable a multitude of optimizations concerning packet processing both in hardware and software. That will demand efficient hardware-software co-design to explore the full potential of CXL.

2.9. Closing Remarks

The emergence of programmable network data plane technologies, particularly SmartNICs, has drastically changed the way network operations and management are handled. By deploying custom networking solutions directly within network devices, operators can have more control and make per-packet forwarding decisions at incredibly high speeds.

The growing interest from both academic and industrial sectors in SmartNICs highlights their potential to revolutionize network performance and efficiency. Leading companies such as Nvidia, Netronome, Intel, AMD, and Xilinx are competing to offer hardware solutions that can offload complex networking tasks from host CPUs. This enhances packet processing capabilities while lowering overall ownership costs.

The evolution of NICs into SmartNICs represents a journey towards greater programmability and specialization, driven by the escalating demands of modern networking environments. However, realizing the full potential of SmartNICs requires grappling with challenges related to programming, debugging, and operating these devices efficiently. Furthermore, understanding the intricacies of SmartNIC architectures and their programming ecosystems is essential for designing and deploying cutting-edge in-network solutions. This chapter provides foundational insights into SmartNIC design principles, hardware architectures, programming languages, and performance considerations. Additionally, it offers a hands-on tutorial featuring state-of-the-art SmartNICs, enabling practitioners to delve deeper into the practical aspects of leveraging this transformative technology.

As SmartNICs continue to evolve and permeate various networking domains, their impact on network performance, scalability, and innovation is poised to be profound. By embracing and mastering SmartNIC technologies, network practitioners can unlock new possibilities for delivering tailored, efficient, innovative, and evolvable networking solutions.

Acknowledgments

This study was partially funded by CAPES, Brazil - Finance Code 001. This work was partially supported by the Innovation Center, Ericsson S.A., and by the Sao Paulo Research Foundation (FAPESP), grant 2021/00199-8, CPE SMARTNESS.

References

[Barach et al. 2018] Barach, D., Linguaglossa, L., Marion, D., Pfister, P., Pontarelli, S., and Rossi, D. (2018). High-speed software data plane via vectorized packet processing. *IEEE Communications Magazine*, 56(12):97–103.

[Bosshart et al. 2014a] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. (2014a). P4: programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95.

[Bosshart et al. 2014b] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. (2014b). P4: programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95.

[Cannarozzo et al. 2024] Cannarozzo, L., Morais, T. B., Lorenzon, A. F., de Souza, P. S. S., Gobatto, L. R., Lamb, I. P., Duarte, P. A. P. R., Azambuja, J. R. F., Lorenzon, A. F., Rossi, F. D., Cordeiro, W., and Luizelli, M. C. (2024). Spinner: Enabling in-network flow clustering entirely in a programmable data plane. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS 2024)*, pages 1–9.

[Chole et al. 2017] Chole, S., Fingerhut, A., Ma, S., Sivaraman, A., Vargaftik, S., Berger, A., Mendelson, G., Alizadeh, M., Chuang, S.-T., Keslassy, I., Orda, A., and Edsall, T. (2017). Drmt: Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 1–14, New York, NY, USA.

[DPDK 2024] DPDK (2024). Internet.

[Guo et al. 2023] Guo, Z., Lin, J., Bai, Y., Kim, D., Swift, M., Akella, A., and Liu, M. (2023). Lognic: A high-level performance model for smartnics. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 916–929, New York, NY, USA.

[He et al. 2023] He, Y., Wu, W., Le, Y., Liu, M., and Lao, C. (2023). A generic service to provide in-network aggregation for key-value streams. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 33–47, New York, NY, USA.

[Intel 2021] Intel (2021). P416 intel tofino native architecture—public version. `https://github.com/barefootnetworks/Open-Tofino"`. accessed July 20, 2023.

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

[Kianpisheh and Taleb 2023] Kianpisheh, S. and Taleb, T. (2023). A survey on in-network computing: Programmable data plane and technology specific applications. *IEEE Communications Surveys Tutorials*, 25(1):701–761.

[Li et al. 2023] Li, H., Berger, D. S., Hsu, L., Ernst, D., Zardoshti, P., Novakovic, S., Shah, M., Rajadnya, S., Lee, S., Agarwal, I., Hill, M. D., Fontoura, M., and Bianchini, R. (2023). Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 574–587, New York, NY, USA.

[Liu et al. 2019] Liu, M., Cui, T., Schuh, H., Krishnamurthy, A., Peter, S., and Gupta, K. (2019). Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 318–333, New York, NY, USA.

[Min et al. 2021] Min, J., Liu, M., Chugh, T., Zhao, C., Wei, A., Doh, I. H., and Krishnamurthy, A. (2021). Gimbal: Enabling multi-tenant storage disaggregation on smartnic jbofs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 106–122, New York, NY, USA.

[Moon et al. 2020] Moon, Y., Lee, S., Jamshed, M. A., and Park, K. (2020). {AccelTCP}: Accelerating network applications with stateful {TCP} offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 77–92.

[Njavro et al. 2022] Njavro, A., Tau, J., Groves, T., Wright, N. J., and West, R. (2022). A dpu solution for container overlay networks. *arXiv preprint arXiv:2211.10495*.

[NVIDIA 2024] NVIDIA (2024). Internet.

[Olteanu et al. 2022] Olteanu, V., Eran, H., Dumitrescu, D., Popa, A., Baciu, C., Silberstein, M., Nikolaidis, G., Handley, M., and Raiciu, C. (2022). An edge-queued datagram service for all datacenter traffic. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 761–777.

[OpenNIC 2024] OpenNIC (2024). OpenNIC.

[Pfaff et al. 2015] Pfaff, B., Pettit, J., Koponen, T., Jackson, E., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., Amidon, K., and Casado, M. (2015). The design and implementation of open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, Oakland, CA.

[(PNA) 2023] (PNA), P. P. N. A. (2023). Internet.

[Ponomarev and Ghose 1998] Ponomarev, D. and Ghose, K. (1998). A comparative study of some network subsystem organizations. In *Proceedings. Fifth International Conference on High Performance Computing (Cat. No. 98EX238)*, pages 436–443.

[Saquetti et al. 2020] Saquetti, M., Bueno, G., Cordeiro, W., and Azambuja, J. R. (2020). P4vbox: Enabling p4-based switch virtualization. *IEEE Communications Letters*, 24(1):146–149.

[Saquetti et al. 2021] Saquetti, M., Canofre, R., Lorenzon, A. F., Rossi, F. D., Azambuja, J. R., Cordeiro, W., and Luizelli, M. C. (2021). Toward in-network intelligence: Running distributed artificial neural networks in the data plane. *IEEE Communications Letters*, 25(11):3551–3555.

[Schuh et al. 2021] Schuh, H. N., Liang, W., Liu, M., Nelson, J., and Krishnamurthy, A. (2021). Xenic: Smartnic-accelerated distributed transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 740–755, New York, NY, USA.

[Swamy et al. 2022] Swamy, T., Rucker, A., Shahbaz, M., Gaur, I., and Olukotun, K. (2022). Taurus: A data plane architecture for per-packet ml. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 1099–1114, New York, NY, USA.

[Viegas et al. 2021] Viegas, P. B., de Castro, A. G., Lorenzon, A. F., Rossi, F. D., and Luizelli, M. C. (2021). The actual cost of programmable smartnics: Diving into the existing limits. In *International Conference on Advanced Information Networking and Applications*, pages 181–194. Springer.

[Wang et al. 2023] Wang, T., Lin, J., Antichi, G., Panda, A., and Sivaraman, A. (2023). Application-defined receive side dispatching on the nic. *arXiv preprint arXiv:2312.04857*.

[Wei et al. 2023a] Wei, X., Cheng, R., Yang, Y., Chen, R., and Chen, H. (2023a). Characterizing off-path SmartNIC for accelerating distributed systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 987–1004, Boston, MA.

[Wei et al. 2023b] Wei, X., Cheng, R., Yang, Y., Chen, R., and Chen, H. (2023b). Characterizing off-path SmartNIC for accelerating distributed systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 987–1004, Boston, MA.

[Wei et al. 2023c] Wei, X., Cheng, R., Yang, Y., Chen, R., and Chen, H. (2023c). Characterizing off-path {SmartNIC} for accelerating distributed systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 987–1004.

[Xavier et al. 2021] Xavier, B. M., Guimarães, R. S., Comarela, G., and Martinello, M. (2021). Programmable switches for in-networking classification. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–10. IEEE.

[Xilinx 2024] Xilinx (2024). Internet.

[Xing et al. 2023] Xing, J., Qiu, Y., Hsu, K.-F., Sui, S., Manaa, K., Shabtai, O., Piasetzky, Y., Kadosh, M., Krishnamurthy, A., Ng, T. S. E., and Chen, A. (2023). Unleashing smartnic packet processing performance in p4. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 1028–1042, New York, NY, USA.

[Yao et al. 2023] Yao, R., Zhang, Z., Fang, G., Gao, P., Liu, S., Fan, Y., Xu, Y., and Chao, H. J. (2023). Bmw tree: Large-scale, high-throughput and modular pifo implementation using balanced multi-way sorting tree. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 208–219, New York, NY, USA.

[Zhang et al. 2023] Zhang, J., Cheng, X., Wang, W., Yang, L., Hu, J., and Chen, K. (2023). {FLASH}: Towards a high-performance hardware acceleration architecture for cross-silo federated learning. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1057–1079.