Performance Trade-offs Analysis of SmartNIC Architectures

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Alessandro Vezzù
20-914-586

Performance Trade-offs Analysis of SmartNIC Architectures

Bachelor Thesis

Scalable Parallel Computing Lab of
Swiss Federal Institute of Technology (ETH) Zurich

Supervision

Prof. Dr. T. Hoefler, M. Khalilov

September 01, 2023

Abstract

In-network packet processing utilizing SmartNICs has emerged as a promising method for enhancing the performance of distributed applications. This approach leverages packet-level parallelism and aligns with the in-network computing trend, which involves integrating energy-efficient processing cores with high-speed NICs. By offloading specific data processing tasks to the networking infrastructure instead of relying solely on the Host CPU, application latency can be reduced, and Host throughput can be maximized by effectively overlapping computation and communication. Over the past decade, several SmartNIC architectures have been developed, accompanied by corresponding APIs for programming them. Off-path SmartNICs incorporate an on-NIC PCIe switch to route packets to a multi-core SoC running an operating system, while on-path SmartNICs integrate cores along the packet communication path with packet manipulation capabilities. To analyze the trade-offs of these two architectures, we conducted a comparative analysis of on-path and off-path SmartNICs. We ported the main components of Caladan, a data plane system that incorporates interference-aware CPU scheduling, to the off-path Bluefield-2 SmartNIC. To assess its performance, we implemented IO and compute-bound tasks on top of Caladan and evaluated their throughput. We then compared the resulting throughputs with those achieved by the same tasks in PsPIN, an on-path RISC-V-based accelerator. We uncover PsPIN's superior performance while acknowledging the increased complexity associated with its application implementation. Conversely, Caladan provides a straightforward pathway for tailoring to distinct use cases and simplifying application development, albeit with potentially lower performance. These findings shed light on the intricate trade-offs and differences between performance and the intricate domains of overall system complexity and complexity of API development in off-path and on-path architectures.

Acknowledgements

I want to express my appreciation to Mikhail Khalilov for his valuable support, useful tips, guidance, and feedback during the writing of this thesis.

Contents

Acronyms and Abbreviations

IP	Internet Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
ARP	Address Resolution Protocol
ICMP	Internet Control Message Protocol
MTU	Maximum Transmission Unit
NIC	Network Interface Card
SmartNIC	Smart Network Interface Card
TX	Transmit
RX	Receive
RDMA	Remote Direct Memory Access
kthread	Kernel Thread
PU	Processing Unit
CRC	Cyclic Redundancy Check
PCM	Performance Counter Monitor
SoC	System-on-Chip
HPU	Handler Processing Unit
LLC	Last Level Cache
HER	Handler Execution Request
CSCHED	Cluster-Local Scheduler
OS	Operating System

Chapter 1

Introduction

Efficient packet processing plays a vital role in distributed computing, as it directly impacts overall system performance. However, relying solely on the Host CPU for data processing faces scalability and efficiency limitations as applications become more data-intensive. Amdahl's Law highlights that the non-parallelizable portions of data processing tasks impose limitations on the overall speedup that can be achieved [4]. However, the challenges of scalability and efficiency in handling the growing demands of data processing are further compounded by the diminishing returns of Dennard [10] and Moore's Law [32] scaling.

Moore's Law, which predicts the continuous increase in transistor density on microchips, has historically led to an exponential growth in computing power and performance. Nevertheless, as Dennard scaling has slowed down in recent years, individual transistors within the CPU are no longer advancing at the same rapid pace as the escalating data intensity of modern applications. This convergence of limitations from Amdahl's Law, Moore's Law, and Dennard scaling collectively hampers the Host CPU's ability to effectively handle the growing demands of data processing, leading to scalability and efficiency challenges.

To address this, a recent advancement has been made by eliminating the CPU from the packet processing path. In its place, dedicated data processors now provide remote direct memory access (RDMA) capabilities, enabling transmission rates of tens of gigabytes per second at sub-microsecond latencies in modern network interface cards (NICs). This significant leap in bandwidth and latency has been achieved through the integration of specialized data processors, allowing for efficient and high-speed data transmission within the network [20].

Modern 400 Gbit/s NICs can receive packets at an astonishing per packet arrival time of around 1-2 nanoseconds [20]. However, placing and reading from L3 cache incurs a higher cost, averaging around 10-15 nanoseconds [20, 31, 21]. Therefore, with the advent of terabits-per-second networks [15], a bottleneck arises in processing the incoming data. The primary issue lies in the fact that packets are indiscriminately stored in main memory, regardless of the nature of their contents. A problem arises as CPU cores are not well-suited for handling messages, given that their microarchitecture is optimized for computational tasks. The time required for thread activation, scheduling, and data movement leads to CPU cores waiting for data processing, resulting in inefficient overall data processing [20].

In recent years, in-network packet processing with smart network interface cards (SmartNICs) has emerged as a promising solution to address the challenges of traditional packet processing architectures [7, 8, 16]. SmartNICs integrate specialized processing cores within high-speed NICs, offering a paradigm shift in packet processing capabilities. SmartNICs enable the utilization of packet-level parallelism, which provides a more fine-grained approach compared to CPU threads. With specialized hardware, data processing is partially offloaded to the networking infrastructure, leveraging this packet-level parallelism. This offloading not only reduces application latency but also maximizes Host throughput by maximizing the degree of computation/communication overlap [28, 42].

SmartNIC architectures have evolved over the past decade, accompanied by APIs for their programming. They can be classified into two types: Off-path and On-path (also known as sideband and Bump-in-the-wire) [28]. Off-path SmartNICs have an on-NIC PCIe switch responsible for routing packets to a multi-core System-on-Chip (SoC) running an operating system like Linux. These off-path SmartNICs offer expanded deployment opportunities as applications can leverage conventional APIs provided by Linux and other libraries, such as DPDK [14], netmap [40], XDP [41], and more. However, off-path offloading can negatively impact application latency and throughput due to routing through the on-NIC PCIe fabric and reliance on the Linux kernel infrastructure [28, 42].

On the other hand, on-path SmartNICs incorporate specialized programmable cores directly on the NIC ingress and egress pipeline with packet manipulation capabilities. Applications need to be programmed using a vendor-specific API to leverage the capabilities of these cores. On-path offloading offers potential performance advantages compared to off-path offloading but requires tailoring applications to the specific offloading API in use to ensure compatibility and maximize performance gains [28, 29, 42].

This work focuses on analyzing two notable systems in the realm of packet processing: PsPIN and Caladan. PsPIN is a RISC-V-based in-network accelerator designed for flexible, high-performance, and low-power packet processing, representing the on-path systems [13]. On the other hand, the Caladan framework integrates interference-aware CPU scheduling to achieve performance isolation and maximize CPU utilization in data center servers [18]. This study uses Caladan as the front-end and back-end for packet delivery, running on the off-path Bluefield-2 NIC [33].

We have ported the key components of the Caladan runtime to the ARM architecture and implemented several applications on top of the user datagram protocol (UDP) API provided by Caladan. In our evaluation, we assessed the throughput and latency of two different kinds of applications: IO-bound and compute-bound. We conducted evaluations using various packet sizes and application threads while executing the corresponding handlers. Furthermore, we also evaluated the performance of the same applications using PsPIN and compared the resulting throughput with that achieved using Caladan. By exploring the efficiency and effectiveness of PsPIN and Caladan, this research sheds light on the capabilities, potential benefits, and trade-offs of packet processing with SmartNICs on off-path and on-path architectures in enhancing overall system performance, reducing latency, and maximizing throughput in data center environments.

Chapter 2

Background

2.1 Off-path and On-path SmartNICs

A Multicore SoC SmartNIC consists of four major parts: computing units with a general-purpose ARM/MIPS multicore processor and accelerators for packet processing and specialized functions (such as encryption/decryption, hashing, pattern matching, compression), onboard memory, a traffic control module for transferring packets between transmit (TX) and receive (RX) ports and the packet buffer, and DMA engines for communicating with the Host [28, 42].

Furthermore, SmartNICs can be categorized as on-path or off-path based on how SmartNIC cores interact with traffic. Off-path SmartNICs like Bluefield-2 [33] and Broadcom Stingray [5] deliver traffic flows to Host or NIC cores using forwarding rules installed on the on-NIC PCIeC switch. These SmartNICs have multicore processors equipped with a full operating system like Linux, making them easily programmable. The operating system running on the SmartNIC operates independently from the Host operating system and can directly process and handle network traffic without involving the Host CPU or kernel. Moreover, it supports zero-copy kernel bypass, allowing it to bypass not just the Host kernel but also the kernel on the off-path SmartNIC [28, 42].

SDKs/APIs like DPDK [14], netmap [40], and XDP [41] can be used to leverage the processing capabilities of the SmartNIC and customize packet handling based on specific requirements. Additionally, there is significant work in developing data plane systems for optimizing operating system (OS) networking for throughput and tail latency. Arachne [39] and Shenango [36] are examples of software primarily operating in the user space. They rely on user-level threading models, specifically green threads, to manage and schedule threads without direct involvement from the operating system's kernel, reducing tail latency and improving programmability. Another system, ZygOS [37], focuses on work stealing to reduce tail latency. Caladan, a data plane system [18], builds on top of these ideas to eliminate network processing and queuing bottlenecks, allowing it to manage interference unperturbed by software overheads or load imbalances.

These data plane systems can effectively use Host OS and kernel bypass on off-path SmartNICs, handling the entire traffic independently from the Host. On the other hand, on-path SmartNICs like LiquidIO [30], PsPIN [13] or Bluefield-3 [34] do not run a full operating system on their cores. Instead, they have handlers offloaded to the NIC cores. In on-path SmartNICs, the NIC cores are part of the NIC ingress and egress pipeline, meaning that incoming traffic always traverses the NIC cores. Handlers offloaded on the cores enable manipulation and preprocessing of incoming traffic. Since no OS is running on the cores, one typically needs to use vendor-specific APIs to develop the handlers. As example, for programming Bluefield-3, the FlexIO SDK [35] can be used, and for PsPIN, their provided interface is utilized [13]. Using these specific APIs results in more complexity and limitations in terms of functionality compared to application development on off-path SmartNICs.

2.2 Caladan: Off-Path Processing and its Stack

Caladan is an advanced data plane system representing the current state of the art [18]. It enhances its capabilities by building upon other systems like Shenango [36] and ZygOS [37]. One notable feature of Caladan is its new CPU scheduler, which achieves significantly lower tail latency and improved throughput compared to standard Linux or Parties [6, 18]. The fact that the entire Caladan runtime operates in user space simplifies the required modifications for the ARM architecture, making them easier to implement. In Section 2.3, we will explore the additional benefits of implementing the data plane in user space, including reducing kernel overhead. Due to these advantages, Caladan was used for off-path processing in this work.

2.2.1 Interference-Aware CPU Scheduling in Caladan

As mentioned earlier in the introduction, the Caladan runtime [18] was used as front-end and back-end for packet delivery, running on the off-path Bluefield-2 NIC [33]. It is specifically designed for low tail latency and high CPU efficiency in data center servers. To accomplish this, Caladan utilizes control signals and policies that prioritize quick core allocation over resource partitioning. The system includes a runtime environment responsible for collecting various control signals, such as CPU usage, memory usage, I/O usage, queuing delays, and request processing times. These signals serve as crucial inputs for Caladan's efficient task scheduling. To effectively respond to these signals and optimize task scheduling, Caladan incorporates two important components. The first component is the centralized scheduler core, also referred to as the `Top-level core allocator`, which actively manages resource contention within the memory hierarchy and among hyperthreads.

The second component of Caladan leverages kernel bypass on the off-path SmartNIC. This involves utilizing a dedicated kernel module, called `Ksched module` that circumvents the conventional Linux Kernel scheduler. By bypassing the traditional scheduler, Caladan achieves the capability of performing microsecond-scale monitoring and precise task placement. `Ksched` provides the scheduler core with essential information, such as global memory bandwidth usage obtained from the DRAM controller. Additionally, it supplies the scheduler core with data on per-core last level cache (LLC) miss rates and notifies it when a task has voluntarily yielded. An overview is given in Fig. 2.1.

To read performance counters from the `Ksched module`, such as the LLC miss rate, the scheduler core initiates a read request through the `ioctl()` interface [2]. Since the `Ksched module` resides in the kernel, shared memory between the kernel module and the scheduler core facilitates direct communication between them. These shared memory regions are depicted in Figure 2.2. The scheduler core writes commands into the shared memory, such as requesting the reading of performance counters, and uses the `ioctl()` interface [2] in conjunction with the `Ksched module` to execute these instructions.

The `Ksched module` is capable of executing three different commands: waking tasks, idling cores, and reading performance counters. This combination of the kernel module and scheduler core enables Caladan to efficiently handle core reallocation and performance counting, even with a large number of concurrent tasks. A more detailed description of the implementation of the scheduling algorithms can be found in the Caladan paper [18, 17].

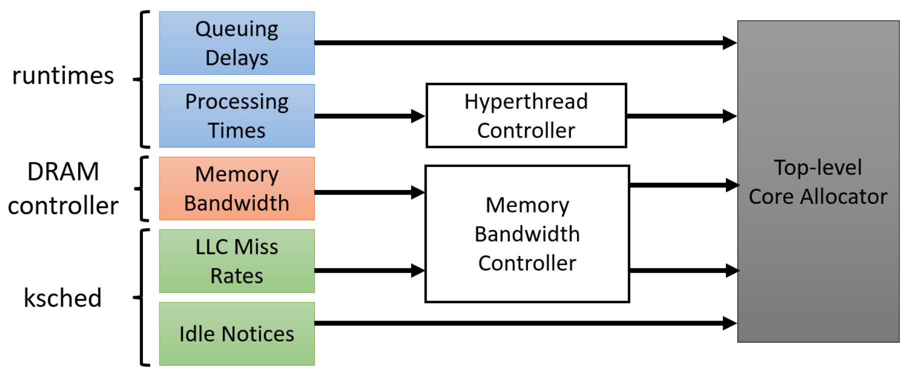


Figure 2.1: The flow of information through Caladan's scheduler [18]

2.2.2 Task Allocation in Caladan

To set up a networking task in Caladan, the first step is to define a main handler and initialize it within the Caladan runtime using a simple C or C++ program. During the initialization process, Caladan automatically creates kernel threads (`kthreads`) dedicated to handling incoming network packets. These `kthreads` play a crucial role in packet processing. When a packet is received, the corresponding `kthread` performs the necessary decapsulation based on whether UDP or transmission control protocol (TCP) was used. The packet is then enqueued in a shared ingress queue, which the main task can access.

This means that the `kthreads` are responsible for executing the relevant network protocols, including handling address resolution protocol (ARP) requests and Internet control message protocol (ICMP) messages, as these protocols are already implemented within Caladan. It is important to note that each task must be assigned at least one `kthread`, which the scheduler assigns to a specific core. Therefore, the number of `kthreads` cannot exceed the available cores. If multiple `kthreads` are created, they work in parallel, distributing the workload across multiple cores to enhance network application performance.

Despite their name, `kthreads` in Caladan are lightweight user-level threads, as the majority of the system runs in userspace, with only the kernel module operating at the kernel level. Even though `kthreads` in Caladan are user-level threads, users still have no direct control over their behavior, as it is predetermined by the Caladan framework. Regarding the task itself, a main thread is spawned along with the corresponding handler and initialized within the Caladan runtime. The main handler provides an interface to access the aforementioned ingress queue, allowing the main process to retrieve the payload of the packet. However, it does not have direct access to the packet header.

When sending packets, the responsibility lies with the main process, which performs the necessary encapsulation. Unlike PsPIN [13], as we will see, Caladan offers APIs for sending UDP and TCP segments. This streamlined approach makes it easier to send packets within the network task. With Caladan, developers only need to focus on the payload itself without having to manage the intricacies of packet encapsulation and decapsulation. Furthermore, these interfaces exclusively utilize the network stack within Caladan and do not rely on the kernel network stack of the operating system [18, 17].

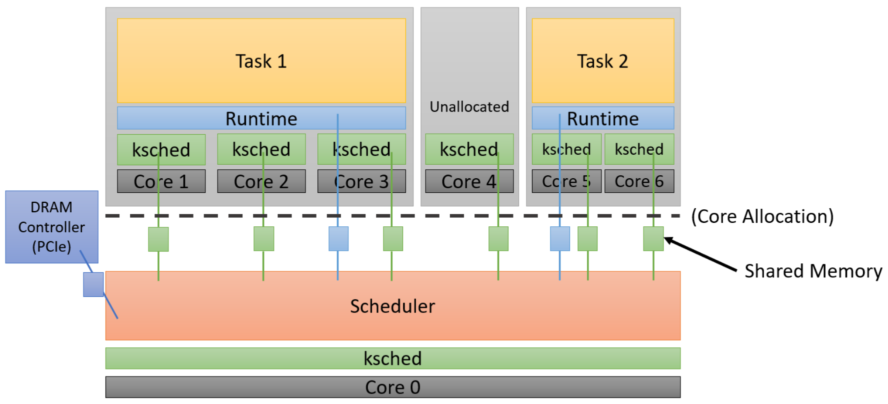


Figure 2.2: Caladan's system architecture [18]

2.2.3 Kernel Overhead Reduction with Directpath

Caladan offers a significant feature known as `Directpath`, built on top of the RDMA-Core library [27]. This library leverages zero-copy kernel bypass using the mlx5 Ethernet poll mode driver to create and access sender and receiver queues directly. When `Directpath` is enabled, the first `kthread` created during the spawning of a new network task will allocate dedicated sender and receiver queues. These queues can be accessed directly by the `kthread` itself for reading or by the main handler for sending operations. This streamlined approach leads to enhanced throughput compared to scenarios where all packets are handled by the `IOKernel`, resulting in improved overall performance [17, 18].

2.3 Caladan's User Space Approach for Data Plane

Caladan's design, residing in the user space, incorporates the implementation of both the network stack and scheduler within this space, resulting in several benefits.

Implementing the network stack in user space brings about a significant reduction in kernel overhead. Caladan efficiently manages read and write operations between user space and the TX/RX queues, utilizing the `Directpath` feature to bypass kernel involvement. This approach minimizes the impact of the kernel on network processing, ultimately improving overall system performance. Furthermore, Caladan's user space data plane empowers users with enhanced control and flexibility over application functionality and behavior. This capability allows for precise customization and optimization of the data plane, tailored to specific use cases or architectural requirements.

Additionally, Caladan situates the scheduler and context switch in user space, which yields multiple advantages. Firstly, this arrangement offers greater flexibility, as the Caladan scheduler is specifically designed to handle interference efficiently, necessitating comprehensive information. Managing and adapting this information becomes more manageable with the scheduler entirely in use space. Secondly, running the scheduler in user space reduces kernel overhead. Context switches between user and kernel modes can incur performance costs. However, the scheduler in user space eliminates the need for frequent context switches, resulting in improved performance and CPU efficiency, as shown in Table 5.1.

2.4 On-path Architecture: PsPIN

PsPIN [13] was selected as on-path architecture as it provides pre-implemented applications [12], offering ready-to-use functionality. Furthermore, PsPIN includes a simulation module that can be easily customized to match the specific specifications and requirements of the Bluefield-2 SmartNICs [33]. Additionally, it achieves a remarkably high throughput of up to 400 Gbit/s [13].

2.4.1 Architecture Overview of PsPIN

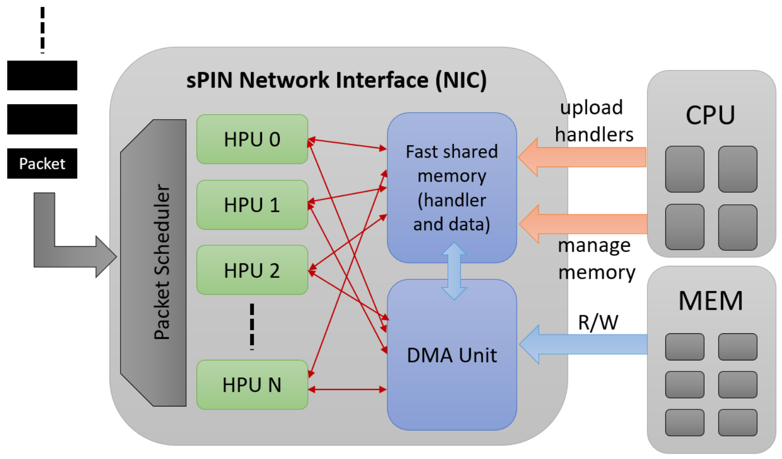


Figure 2.3: sPIN Architecture Overview [20]

For on-path offloading, PsPIN [13] was utilized, which is an implementation of the portable programming model sPIN [20]. The fundamental concept of this model is packetization, which allows programmers to define headers, payloads, and completion handlers (kernels) that operate on exposed packets. These handlers are offloaded to handler processing units (HPUs), which can be thought of as NIC cores.

In the sPIN network layer, all the necessary information to identify and route a message into memory is included in the first packet, known as the header packet. The header information is exclusive to the first packet and is not present in the payload packets. The user-defined header handler is executed on the header packet to extract the essential details. Subsequent packets, except for the last one, trigger the payload handler, while the completion handler is called for the final packet. Importantly, packet order or arrival does not need to be maintained.

The scheduling of handlers on HPUs is managed by a simple runtime system. Each handler has access to shared memory that remains persistent throughout the message's lifetime, enabling communication among handlers. The Host operating system handles the management of HPU memory, while the user-level application adheres to the conventional control/data-plane separation [20]. In PsPIN, the HPUs are organized into processing clusters. Each HPU is implemented as a RISC-V core, and each cluster is accompanied by a scratchpad memory known as L1 memory, providing single-cycle access. Although other clusters theoretically have access to this L1 memory, memory access in such cases incurs higher latency. Apart from the L1 memory, PsPIN features three off-cluster memories: the packet buffer, the handler memory, and the program memory [13].

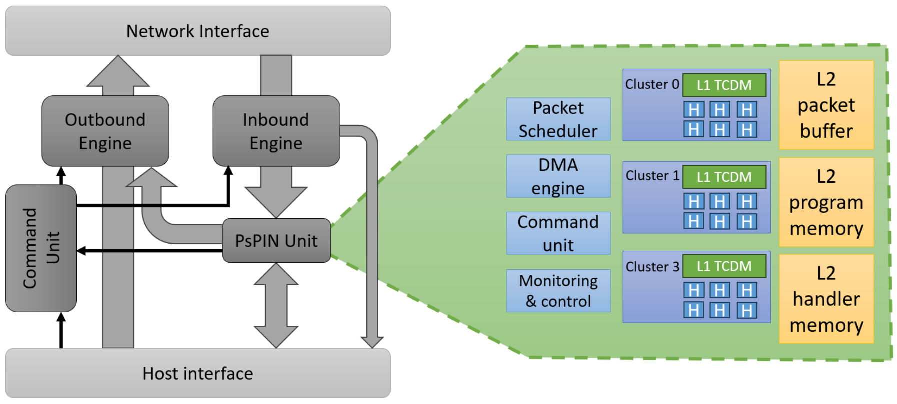


Figure 2.4: NIC model and PsPIN architecture overview [13]

2.4.2 Packet Processing Pipeline of PsPIN

Once the code and data for the handlers have been offloaded, the Host constructs an execution context that includes various pointers necessary for packet processing. This execution context is then transferred to the NIC and utilized by the NIC's inbound engine to route packets to PsPIN. Upon receiving an incoming packet, the NIC's inbound engine processes it by typically copying the packet data into Host memory. To determine which packets should be processed by PsPIN, the engine matches packets to PsPIN execution context. If a match is found, the packet is forwarded to the PsPIN unit. If no match occurs, the packet is copied to the Host memory.

In the next step, the packets are copied into the L2 packet buffer, and a handler execution request (HER) is sent to PsPIN's packet scheduler. The scheduler then selects the processing cluster responsible for processing the new packet. Within a cluster, a cluster-local scheduler (CSCHED) initiates DMA copy operations to move the packets from the L2 packet buffer to the L1 memory. It also selects an idle HPU where the handler for the packets available in L1 can be executed. Once packet processing is complete, a notification is sent back to the NIC, allowing it to update its view of the packet buffer.

For the reverse operation, sending data, the sPIN API provides an RDMA-put operation. This involves sending a blocking NIC command to the NIC's outbound engine for data transmission. The NIC can send data from either the L2 packet memory or the L1 memories or directly transmit data from Host memory [13].

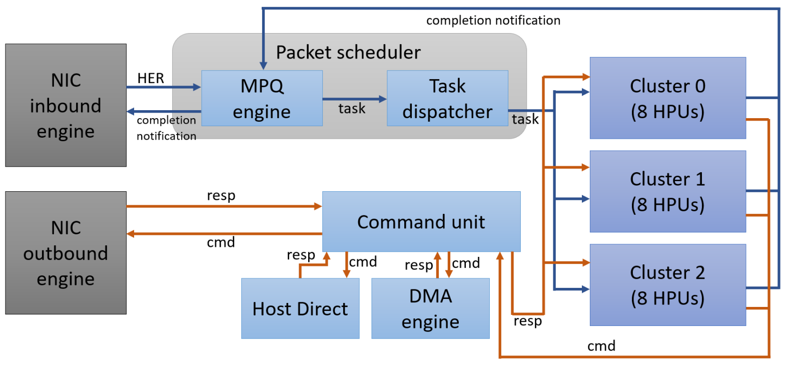


Figure 2.5: PsPIN control path overview [13]

Chapter 3

Porting Caladan's Data Plane System to ARM Architecture

3.1 Changes to Caladan

In this work, the Bluefield-2 SmartNICs [33] were utilized as off-path SmartNICs. These SmartNICs incorporate ARM DPUs. As Caladan was originally developed for the x86 architecture, significant adjustments were made to make it compatible with the ARM architecture for execution on Bluefield DPUs.

Firstly, the RDMA-Core library [27] needed to be updated to ensure compatibility with the Bluefield-2. The original version was unsuitable for the ARM DPUs, so modifications were made to enable proper functionality. Consequently, version 44 was used instead of version 32. Additionally, a small inline assembly in the RDMA-Core library had to be adjusted since ARM's instruction `pause` is called differently.

Next, all functionalities related to the performance counter monitor (PCM) library [9] from Intel were removed, including instructions to read CPU states. Moreover, any other inline assembly instructions used to read the CPU state, unrelated to the PCM library, had to be adjusted for ARM. These instructions involved reading the CPU ID or the cycle counter register. Another aspect of the adaptation process was rewriting certain functionalities implemented initially in x86 assembly, such as the Internet protocol (IP) checksum [23] and cyclic redundancy check (CRC) [38]. Both checksums were rewritten in C, with the IP checksum being natively implemented and the CRC using the zlib1g library [3].

However, the most challenging aspect of the adaptation was the context switch. Caladan had its own context switch implemented in x86 assembly, which needed to be reworked for ARM assembly. This involved several tasks: identifying all registers that must be saved for a context switch on the ARM architecture, implementing the context switch in ARM assembly, and including all additional checks and function calls within the context switch (as in the original x86 implementation). Additionally, since different registers are used in ARM, the thread structure implemented in Caladan needed to be adjusted. Adjustments because of ARM calling conventions were also necessary. In x86, the return address is stored on the stack, whereas in ARM, it is stored in a register. Therefore, the thread initialization had to be changed to ensure the correct address was held in the return address register. Moreover, since there is no return address on the stack in ARM, the stack alignment needed to be corrected by changing the initialization address of the stack to ensure proper 16-byte alignment.

In addition to the thread structure, adjustments were also made to the per-thread variable declaration. In the original Caladan implementation, per-thread variables were placed in the .perthread section. However, this syntax for such a declaration does not exist for ARM, as it is compiled differently. Instead, per-thread variables were declared as thread variables without specifying the section.

The last changes were made in the `Ksched module` support. In the kernel module, all CPU state reads specific to x86 had to be removed. The function `mwait`, which is only available for x86 and allows efficient idling of cores [22], was also removed and replaced with busy waiting. Furthermore, the driver initialization was adapted. The `Ksched module` assumes the presence of a CPU idle driver that can be hijacked when loading the module [17]. However, since the DPUs do not have such a driver, the hijacking part was removed and replaced with registering the driver. Similarly, the unjacking process was replaced with deregistering the driver.

Here, a limitation of the ported version arises. Registering the driver alone is insufficient to enable the functionalities of the `Ksched module`. Implementing proper initialization of a CPU idle driver was not feasible within the scope of this thesis. Consequently, the kernel module is unable to execute any commands sent from the Caladan runtime, and therefore all corresponding `ioctl()` [2] calls were removed from the Caladan implementation.

Overall, the adjustments made to Caladan for the ARM architecture involved updating libraries, modifying inline assemblies, rewriting functions, implementing a context switch for the ARM architecture, adjusting the thread implementation, and removing all unsupported functionalities. These adaptations were necessary to ensure smooth execution and compatibility on the Bluefield DPUs.

File	Lines
rdma-core/providers/mlx5/dr_send.c	1
inc/asm/ops.h	9
inc/asm/chksum.h	26
inc/base/thread.h	4
inc/runtime/preempt.h	18
ksched/ksched.c	30
runtime/sched.c	13
runtime/defs.h	53
runtime/switch.S	281

Table 3.1: This table provides an overview of the amount of rewritten C and ARM assembly lines, excluding any deletions. The total number of lines that have been rewritten amounts to 435.

Chapter 4

Porting Applications for Off-path Processing

To gain a deeper understanding of individual system performance, we conducted measurements on the throughput of two types of tasks: compute-bound and IO-bound tasks. Compute-bound applications are constrained by computational costs, allowing us to assess the CPU's ability to handle incoming packets that require significant computation. On the other hand, IO-bound tasks were also tested, as they do not involve any expensive computational processes. The purpose of these tasks is to evaluate the system's memory management, including latency for memory reads, writes, and RDMA operations. In order to compare the performance of these tasks on Caladan [18] and PsPIN [13], we needed to implement the applications for both systems. To achieve this, we utilized the pre-existing examples available in the GitHub repository of PsPIN [12] and ported them to the Caladan framework using the provided interfaces.

4.1 Data processing in Caladan

For the benchmarks in Caladan, the applications were ported using UDP and TCP interfaces, which differs from PsPIN's implementation, which utilizes handlers for packet processing. Notably, the time taken for connection establishment, including the TCP handshake and the ARP request to obtain the MAC address, was not included in the measurement. The focus was specifically on measuring the performance during data transmission and processing, excluding the initial setup time.

For measuring the time, the benchmarks were conducted using the `gettimeofday (struct timeval *restrict tv, struct timezone* tz)` function from the standard C library [1]. It is important to note that all time measurements were made in microseconds. Furthermore, in the absence of any information regarding the Maximum Transmission Unit (MTU), a default MTU value of 9000 bytes was selected. Also, in each benchmark, precisely 1000 packets were sent, either from the client to the server or from the server to the client, depending on the specific application being tested. The decision to send 1000 packets was deliberate, as sending more packets resulted in packet drops. One final point to note: when referring to a packet, we specifically address its payload. For instance, if the application sends a packet containing 1000 bytes, it means a payload of 1000 bytes is being transmitted. However, some additional bytes will be sent along with it due to the packet header.

4.1.1 Empty Handler

To have a baseline, allowing us to understand how throughput behaves in a very simple scenario and to assess the applications performance within the systems, we implemented an empty handler. Both client and server processes are provided with the buffer size as an input parameter. After receiving the buffer size, the client starts transmitting the packets to the server, with each packet being the same size as the buffer size. Upon receiving the first packet, the server initializes the first timestamp. Subsequent packets are received, and their buffer sizes are accumulated until the final packet is received, which triggers the second timestamp. Finally, the server computes the throughput by dividing the total number of received bytes by the measured time.

4.1.2 Empty Handler with different MTUs

Since the empty handler represents a simplistic scenario, we were also interested in examining how throughput and latency behave when altering the MTU size. Therefore, in this benchmark, we slightly modify the empty handler by including the MTU as an input parameter for both the client and server processes instead of using the buffer size. Consequently, the buffer size was set to be equal to the chosen MTU value. For example, if an MTU of 3000 bytes was selected, the buffer size was also set to 3000 bytes. Apart from these adjustments, the benchmark followed the same procedures as discussed in the section 4.1.1.

4.1.3 IO-bound Benchmark Applications

PingPong

The purpose of this IO-bound application is to measure the time it takes for a packet to be sent and for a corresponding response to be received. In order to accomplish this, both the server and client processes are provided with the buffer size as an input parameter. The client generates a timestamp and sends a packet, which has the same size as the buffer size, to the server. The server then receives the packet and sends back an identical one as a response. Upon receiving the response, the client takes a second timestamp, and the resulting latency measurement is recorded.

CopyFromHost

The second IO-bound application we ported was CopyFromHost, which served as a means to investigate how RDMA read affects throughput. The application operates as follows: the server and client receive a buffer size as input. In the second step, the server initiates an RDMA connection to the Host. It's worth noting that in the case of using multiple application threads, a separate RDMA connection was established for each created thread, enabling concurrent RDMA read operations.

Next, the server performs an RDMA read from the Host Memory with a probability of 0.5. If an RDMA read is not performed, the server simply accesses local memory. The read data is then sent to the client, where throughput measurements are conducted. Similar to the empty handler scenario, the first packet serves as a trigger for the first timestamp on the client side and the buffer size of the received packets is accumulated. Once the last packet is received, the second timestamp is recorded.

CopyToHost

This IO-bound task can be considered the inverse of the CopyFromHost application. Instead of investigating how RDMA read influences the throughput, we focus on how RDMA write affects it. Similar to the previous benchmarks, the server and client receive the input buffer size as parameters. The server initiates an RDMA connection to the Host, and in the case of using multiple application threads, separate RDMA connections are established for each created thread.

Once the setup is completed, the client sends packets to the server. For each packet received, the server utilizes RDMA write to copy the data to the Host memory, except for the first packet. Also here, the server adds up the buffer size of all packets received from the client. The first packet serves as a trigger for the timestamp recording. The second timestamp is recorded when the last packet is received, concluding the measurement.

Filter

To assess the impact of a small additional and constant workload coupled to a IO-bound task, we ported the filter application. In this particular benchmark, the client and server processes are provided with the buffer size as an input parameter. Initially, the server initializes a hash table where entries are set to zero or some random number greater than zero. Subsequently, the client sends a packet to the server, with the packet size matching the received input size. This packet is initialized with a random number. The arrival of the first packet marks the starting point for the timer.

For each subsequent packet, the server calculates the hash value of the first 8 bytes of the received payload, utilizing the Jenkins hash function [25]. This calculated hash value is then used for a lookup in the hash table. If the lookup value in the table is zero, the server copies the buffer via RDMA to the Host memory. Otherwise, the packet is filtered out, meaning nothing happens. Independent of whether the packet was filtered out, the received buffer size was included in the throughput calculation. As in the CopyFromHost and CopyToHost applications, in case of using multiple application threads, each thread has its own RDMA connection to the Host. Once the server receives the final packet, it takes the second timestamp and calculates the throughput.

4.1.4 Compute-bound Benchmark Applications

Aggregate

To assess compute-bound tasks, we ported three applications, one of which is the aggregate application. The functioning of the aggregate application is as follows: upon receiving a buffer size as an input parameter, the server process initializes a shared memory region, which corresponds to a volatile declared memory region having a size of 8 bytes. This shared memory can be accessed by different threads in a multi-threaded version of the server.

After the initialization of the shared memory region, the client starts to send packets to the server. Each packet has the size of the received buffer size, where each element is an integer. As with previous descriptions, the timer starts with the arrival of the first packet. The server threads then sequentially sum up all elements within the buffer locally. The resulting sum is added to the shared memory region using the provided atomic addition implementation from Caladan, as shown in Fig. 4.1.

Once the server receives the final packet, it writes the shared memory via RDMA to the Host memory and captures the second timestamp to mark the operation's completion. It's important to note that the shared memory is only copied once to the Host memory and not by every thread in case of using multiple threads. The resulting throughput measurement, which consists of the time taken and the accumulated buffer sizes received on the server side, is subsequently stored for evaluation.

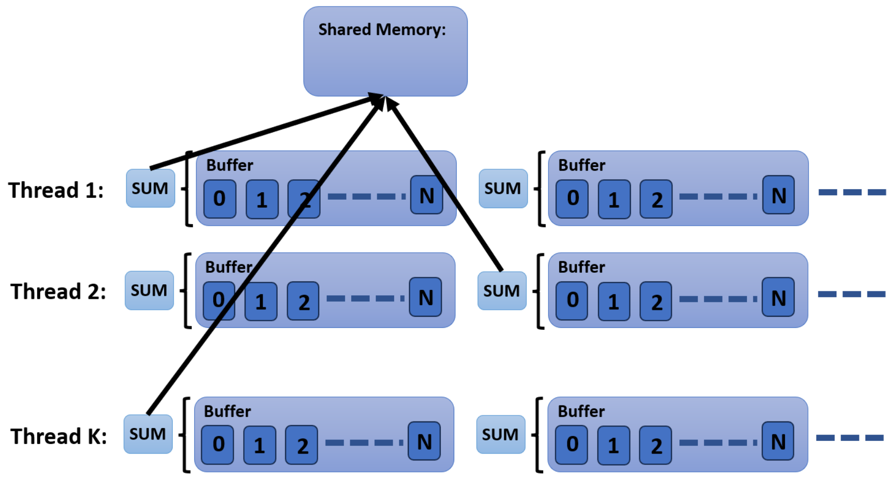


Figure 4.1: Overview of the aggregate operation

Reduce

As a second compute-bound task, we also ported the reduce application. In this case, the process receives a buffer size as an input parameter and initializes a shared memory region with a size that matches the received buffer size. Apart from the size of the shared memory, the reduce application has one key difference compared to the aggregate application. Instead of summing the elements locally, the server threads sequentially read each element from the buffer and add it to the corresponding region in the shared memory. Specifically, the i-th element of the buffer is consistently added to the i-th element of the shared memory, as illustrated in Figure 4.2. Similar to the previous scenarios, all shared memory accesses are handled using atomic addition. Apart from these variations, all other benchmark aspects remain identical to the aggregate benchmark described in section 4.1.4.

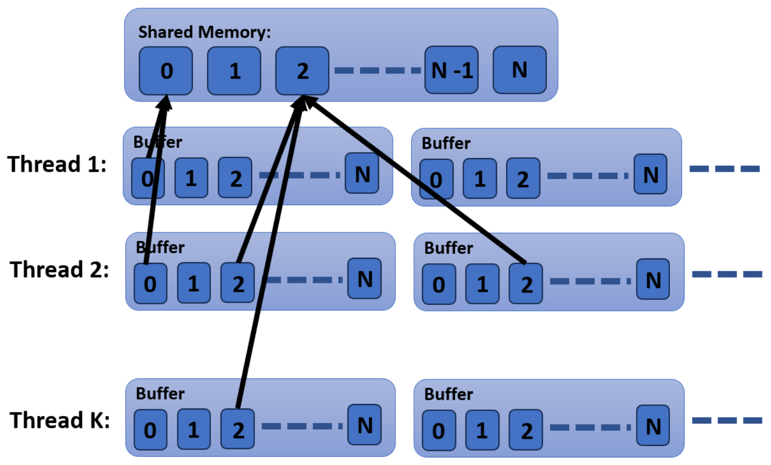


Figure 4.2: Overview of the reduce operation

Histogram

In this benchmark, we shift our focus from the reduce operation to a histogram operation. The fundamental setup remains similar to that of the reduce application but with a difference in the memory access pattern of the shared memory. Instead of summing the received values, the server now increments a counter for each specific number encountered. Each unique number is associated with its dedicated counter, enabling the construction of a histogram based on the received numbers. These counters are also implemented as shared memory, facilitating access by multiple threads in a multi-threaded version.

Similar to the reduce application, the increment of counters utilizes the provided atomic increment implementation of Caladan. Besides this modification, the benchmark follows the same procedures outlined in the earlier reduce application. The timer starts with the arrival of the first packet, and the server threads sequentially process each element of the received buffer to update the corresponding counters in the shared memory. Once all packets have been received and processed, the server writes the final histogram data to the Host memory via RDMA and captures the second timestamp to mark the operation's completion.

Benchmark Applications	Lines
Emtpy Handler with TCP	194
Emtpy Handler with UDP	187
PingPong with TCP	134
PingPong with UDP	128
CopyFromHost	263
CopyToHost	226
Filter	356
Reduce	221
Histogram	218
Aggregate	218

Table 4.1: This table provides an overview of the total number of lines of C code written for the benchmark applications.

Chapter 5

Results

5.1 Execution of the Benchmarks

5.1.1 Execution of Caladan Benchmarks on the Host

The benchmarks were conducted in the following manner: A Python script was utilized to initiate all the necessary processes and pass the required input parameters. Each process allocated a single `kthread`, and `Directpath` was employed for all benchmarks. To ensure proper packet transmission, all client processes were bound to a specific port distinct from the server's port. For each input parameter, every application was executed 100 times. For instance, if a buffer size of 1024 bytes was specified, the application would be executed 100 times with a buffer size of 1024 bytes. In case of a failed attempt, it was repeated until a successful outcome was achieved. Only the successful attempts were considered for the evaluation of the results.

Notably, the processes were aware of the exact number of packets they should receive. If any packets were missing, the benchmark was deemed unsuccessful, ensuring no packet loss for the measured values. After executing an application 100 times, the arithmetic mean was calculated for latency or the harmonic mean, depending on whether throughput was being measured [19]. The resulting mean value was then saved, and the entire process was repeated for the next set of input parameters until all tests for all input parameters were completed.

5.1.2 Execution of Caladan Benchmarks on Bluefield-2 DPUs

The benchmarks were conducted following a similar procedure as on the Host, but with a notable difference: the server and client processes were allocated on different

SmartNICs. Additionally, certain benchmarks were scaled by varying the number of `kthreads` and application threads used. The number of `kthreads` is always equal to the number of application threads, i.e. when the number of threads is specified, it always refers to both. The remaining steps of the procedure remained consistent with the description provided in the previous section.

Additionally, we conducted cycle measurements of different operations in Caladan. To measure the cycles, we directly read the cycle count from the `pmccntr_el0` register using the assembly instruction `mrs` [11]. As reading the `pmccntr_el0` register from user space is not permitted by the operating system, we utilized the armv8_pmu_cycle_counter_el0 module [24] to overcome this restriction.

5.1.3 Execution of PsPIN Benchmarks

For PsPIN a simulation model was used [12]. The inbound engine of this model takes a trace of packets as input and injects them into PsPIN at a specified rate [13]. This rate, denoted by the input delay parameter, was set to 650 when executing the benchmarks, aiming to achieve a maximum throughput of 100 Gbit/s for the empty handler. This choice represents the highest attainable throughput considering the setup used and taking into account the limitations of the SmartNIC.

To further enhance the experimental configuration, two clusters were selected, each comprising four HPU cores. Accordingly, we adjusted the values of `NUM_CLUSTERS` and `NUM_CORES` to 2 and 4, respectively. Additionally, to accommodate packet sizes up to 8192 bytes, we modified `DEFAULT_NO_MAX_PKT_SIZE` to 16384. The execution of the benchmarks followed a similar process as in Caladan. A Python script passed the necessary input parameters and initiated the simulation. Since it was a simulation, each benchmark was executed only once, and a total of 32 packets were transmitted. Feedback latency and throughput were automatically measured by the simulator itself, when the application was completed.

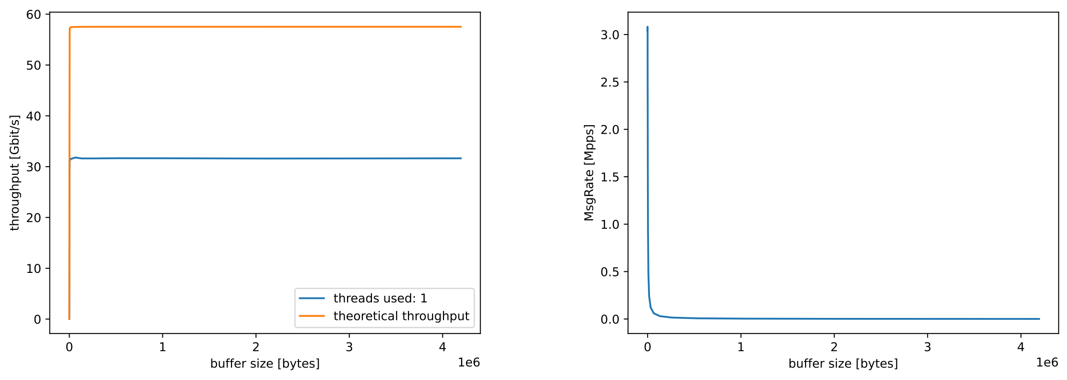
5.2 Benchmarks for Caladan on Host

This section comprises benchmarks conducted on the Host machine. We began with the simplest benchmark scenario since no porting of Caladan was required on the Host. These Host benchmarks were not intended for comparison with other systems, especially not with PsPIN. Instead, their primary purpose was to test the Caladan setup on the Host machine and verify whether the packets were sent through the SmartNICs, while also ensuring that the measured rate aligns with the hardware setup. Consequently, we only utilized specific configurations and did not perform all benchmarks for this particular testing phase.

5.2.1 Throughput of Emtpy Handler

Results for TCP

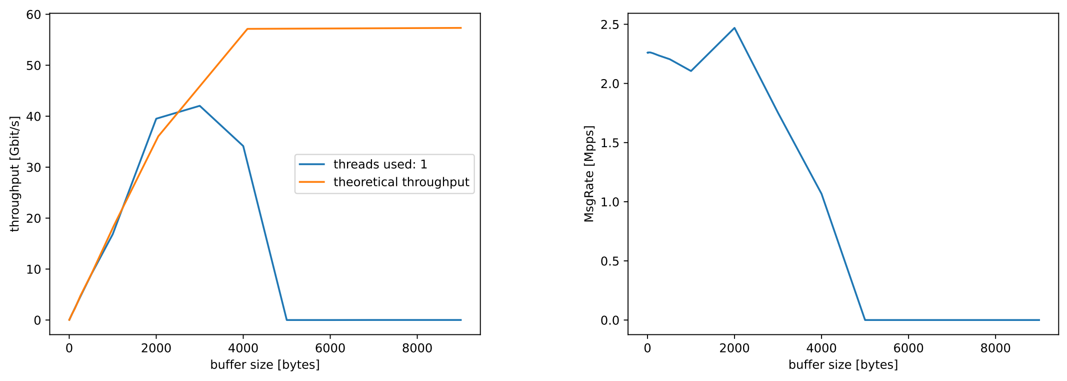
This operation was primarily affected by the additional overhead introduced by TCP. When comparing it with the results obtained with UDP, see Fig. 5.2a, it is evident that the time spent in the TCP protocol significantly impacts the throughput. Nevertheless, it is very stable; the throughput remains almost equal.



(a) Throughput **(b)** Message rate

Figure 5.1: Throughput and message rate for only receiving packets without any computation using TCP. The maximum theoretical throughput of around 57 Gbit/s was derived from the PCIe on the motherboard through which the SmartNICs were connected.

Results for UDP



(a) Throughput **(b)** Message rate

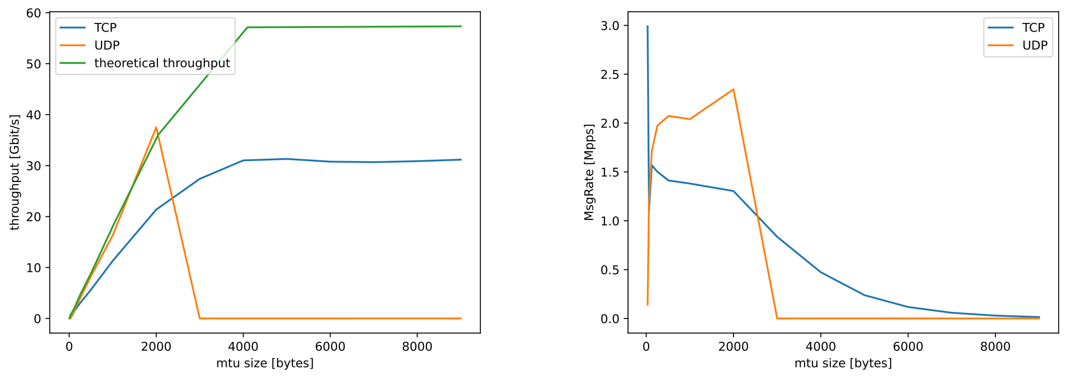
Figure 5.2: Throughput and message rate for only receiving packets without any computation using UDP. The maximum theoretical throughput of around 57 Gbit/s was derived from the PCIe on the motherboard through which the SmartNICs were connected.

In this scenario, the achieved throughput is significantly better than when using TCP. For certain buffer sizes, we were able to approach the maximum theoretical throughput. Further investigation revealed that by using multiple threads, including both application threads and kernel threads, a throughput of approximately 54 Gbit/s could be reached, which is very close to the theoretical maximum throughput. Notably, the significant drop in the graph can be attributed to the execution of the server and client on the same Host. However, by separating them on two different SmartNICs and utilizing multiple threads, this drop was effectively eliminated.

In summary, the conventional Host setup with TCP is very stable but is affected by significant overhead, whereas UDP is much faster but inefficient in handling large packets.

Results using different MTUs

We did not anticipate significant changes in the results when using different MTUs, which was generally the case. However, there is a noticeable difference, particularly for UDP, where the graph collapses much earlier when using an MTU size other than 9000 bytes for all packets. In Figure 5.2a, it can be observed that the drop occurs at a buffer size of approximately 5000 bytes, whereas with a variable MTU size, the drop happens at a buffer size of around 3000 bytes. The reason behind this behavior is currently unclear to us.

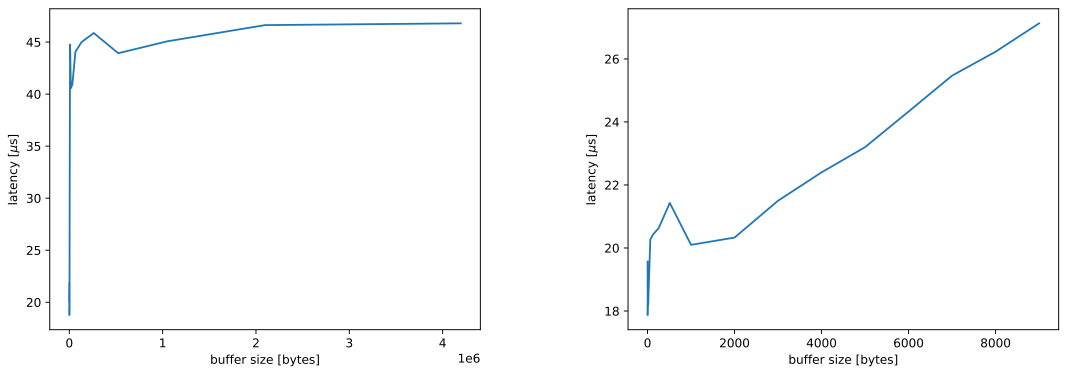


(a) Throughput **(b)** Message rate

Figure 5.3: Throughput and message rate for receiving packets without any computation using different MTUs. The maximum theoretical throughput of around 57 Gbit/s was derived from the PCIe on the motherboard through which the SmartNICs were connected.

5.2.2 Latency of PingPong

In the case of the PingPong operation, we expected to achieve low latency due to the inexpensive computations involved. It is interesting to observe how the TCP protocol impacts the latency of the PingPong operation. With TCP, we experience nearly double the latency compared to UDP.

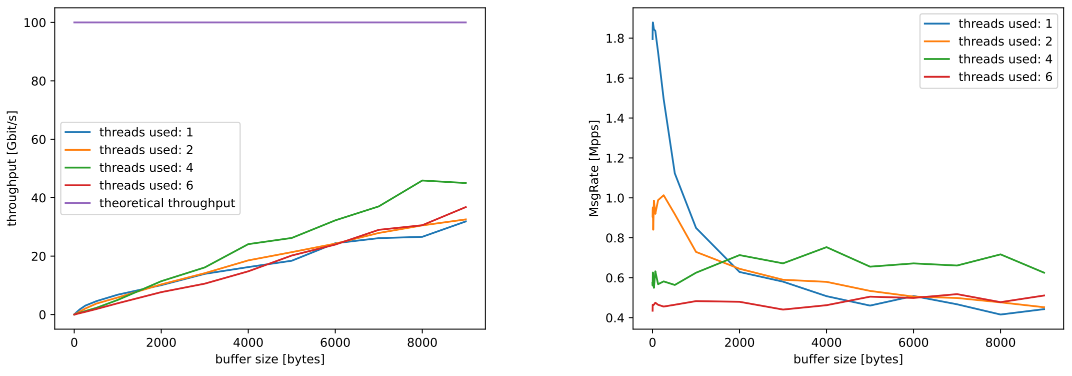


(a) Benchmark with TCP **(b)** Benchmark with UDP

Figure 5.4: PingPong with different buffer sizes

5.3 Benchmarks for Caladan on Bluefields

5.3.1 Throughput of Emtpy Handler



(a) Throughput **(b)** Message rate

Figure 5.5: Throughput and message rate for receiving packets without any computation using UDP. The theoretical throughput of 100 Gbit/s was determined based on the specifications and capabilities of the SmartNIC. It represents the maximum potential throughput supported by the SmartNIC and the 100 Gbit/s cable connection.

The empty handler measurement aims to provide a baseline for the maximum achievable throughput of SmartNICs using a multithreaded server and multiple `kthreads` within the Caladan framework. It can be observed that the peak of the graph is around 50 Gbit/s. Additionally, it was found that the throughput of a single thread sending packets also reached 50 Gbit/s. Interestingly, employing a multithreaded client with a higher packet-sending throughput does not alter the result. Therefore, it becomes evident that the bottleneck lies in packet processing on the server side.

It is worth noting that performance diminishes when utilizing more than four threads. In terms of packet reception, multiple shared resources necessitate the use of a mutex. One such shared data structure between the `kthreads` and application threads is the ingress queue. After packet decapsulation, a `kthread` locks the queue and executes the corresponding protocol, such as UDP. The queue remains locked until the protocol termination [17]. This process, coupled with the fact that 12 threads in total attempt to access the queue and other shared resources, leads to the conclusion that, in the case of six `kthreads` and six application threads, we encounter a bottleneck caused by the overhead associated with multithreading.

In summary, the overhead induced by Caladan, including synchro-

nization of shared resources, limits the throughput to 50 Gbit/s, which falls short of the theoretical maximum throughput of 100 Gbit/s.

5.3.2 Results of IO-bound Benchmark Applications

Latency of PingPong

In the case of the PingPong application, we expected low latency. Consequently, we used only one application thread and one `kthread` since the server receives and sends back only one message. When comparing the latency of the PingPong benchmark on the Host with that on the SmartNIC, there is some variation. The small difference aligns with our expectations, as the PingPong protocol does not require any expensive computation. Furthermore, when executing the protocol on the Bluefields, there is no copy to the Host memory, which results in slightly lower latency.

In summary, the removal of additional latency caused by copying to the Host memory improves performance, surpassing the results of the PingPong benchmarks on the Host.

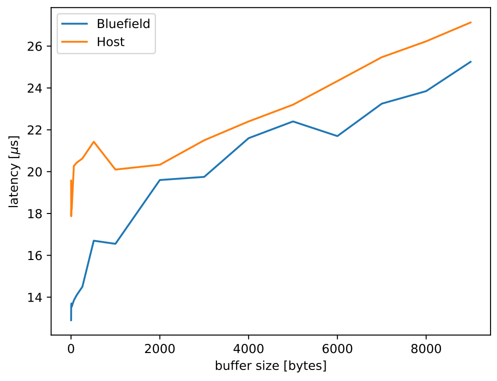
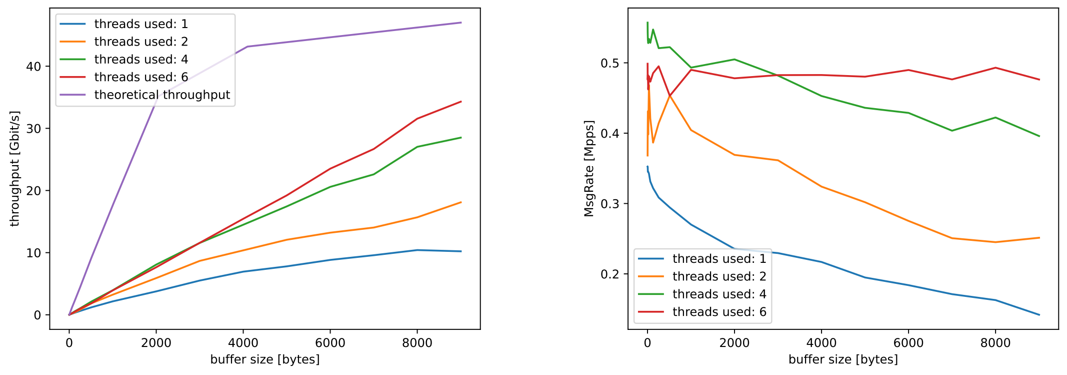


Figure 5.6: Latency of PingPong with different buffer sizes using UDP.

Throughput of CopyFromHost

The throughput for the CopyFromHost operation was slightly lower than the throughput of the CopyToHost operation, even though the CopyToHost operation requires more time, as shown in Figure 5.13b and Figure 5.13a. The reason for this difference lies in the significant disparity between the computational effort involved in receiving a packet and performing a CopyFromHost operation. In the CopyFromHost operation, we measure the throughput on the client side to determine the efficiency of receiving packets when some of them need to be read from the Host memory. The cycle measurements confirm that receiving a packet is much less computationally intensive than sending it in a CopyFromHost operation, including the RDMA read. Consequently, the client ends up waiting for the next packet, leading to a lower throughput. However, by reducing the frequency of RDMA reads, we can observe an overall increase in throughput.

In summary, the significant difference in computational effort between receiving a packet and performing a CopyFromHost operation results in reduced throughput on the client side.



(a) Throughput **(b)** Message rate

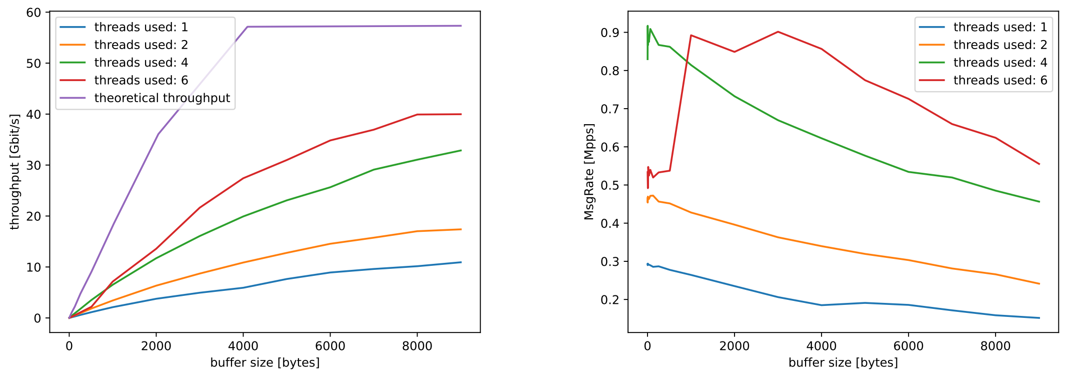
Figure 5.7: Throughput and message rate of the CopyFromHost operation using UDP. The throughput of the RDMA read operation can be regarded as an upper bound for the maximum theoretical throughput.

Throughput of CopyToHost

The result of the CopyToHost operation was as expected. The only additional work involved in this operation was the RDMA write to the Host memory. Overall, the operation is similar to the filter operation, and therefore, a similar performance could be anticipated. In contrast to the filter operation, where only specific data received from the client is written to the Host memory, every received packet is written to the Host memory in the CopyToHost operation.

This additional processing time eliminates the remaining overhead observed in the filter operation in Fig. 5.9a when using six threads. Now, with every additional thread, the throughput increases. It is also evident that the overhead is eliminated relatively early, similar to the aggregate operation, see Fig. 5.10a. Further investigation into the cycle measurement confirms that the throughput is primarily determined by the latency of the CopyToHost operation, thus validating the elimination of overhead.

In summary, while we do incur additional latency through the RDMA write operation, it effectively eliminates multithreaded overhead, enabling efficient utilization of 6 threads.



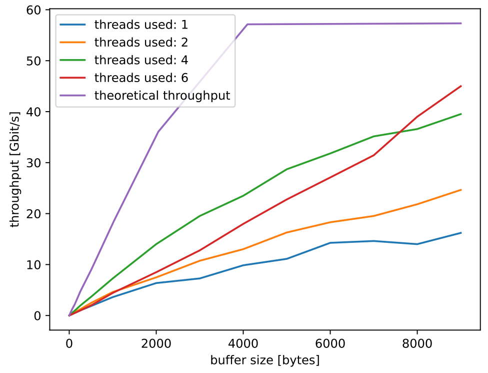
(a) Throughput **(b)** Message rate

Figure 5.8: Throughput and message rate of the CopyToHost operation using UDP. The maximum achievable throughput is determined by the throughput of the RDMA write, as it sets an upper limit for the overall process.

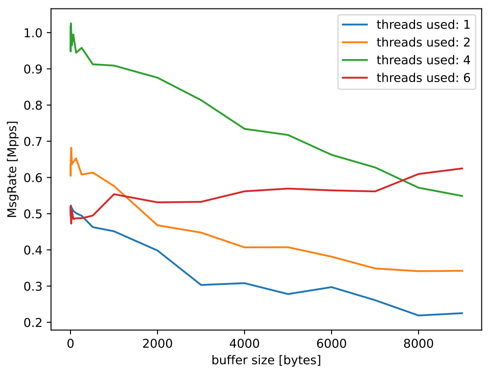
Throughput of Filter

The filter operation itself is independent of the buffer size, as it always takes the first two elements to calculate the lookup value. In the measurement presented in Fig. A.1, it can be observed that RDMA achieves a throughput of approximately 56 Gbit/s. Consequently, it can be deduced that the overall performance of the filter operation is mainly limited by packet processing. Additionally, we must consider the time required for the filter operation and the RDMA write, which results in a slightly lower throughput compared to our empty handler, see Fig. 5.9a and Fig. 5.5a. It is also interesting to note that for the largest buffer size, the overhead caused by multithreading appears to be eliminated.

In summary, the RDMA writes to the Host and filtering calculations cause a slight reduction in throughput compared to the empty handler.



(a) Throughput



(b) Message rate

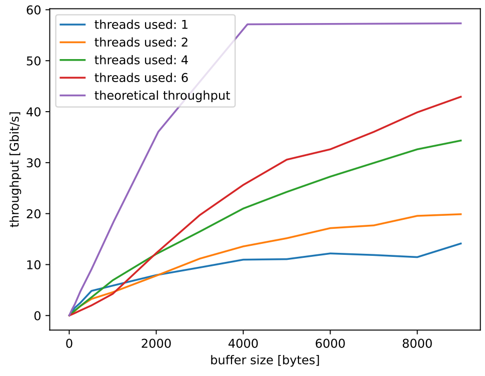
Figure 5.9: Throughput and message rate for the filter operation using UDP. The maximum achievable throughput in the filter operation is determined by the throughput of the RDMA write, as it sets an upper limit for the overall process.

5.3.3 Results of compute-bound Benchmark Applications

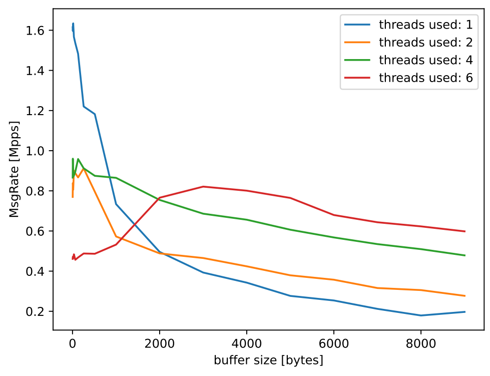
Throughput of Aggregate

In this benchmark, we observe that the more computationally expensive operations eliminate the overhead caused by multi-threading relatively early, and the overall throughput scales with the number of threads after a buffer size of 2000 bytes. As the application threads now operate dependent on the buffer size, the overhead gradually diminishes with increasing buffer size. Consequently, the throughput of the threads also scales with the number of threads earlier compared to the filter operation, see Fig. 5.9a. However, according to the cycle measurement Fig. A.1, the overall throughput is still predominantly influenced by the time required for packet processing, specifically the encapsulation of the packet and its enqueuing in the ingress queue. Although the latency difference between the aggregate operation of a single packet and the encapsulation process and enqueuing of the packet is minimal, the latency of the aggregate operation remains consistently lower for all buffer sizes.

In summary, summing the elements of the buffer in local memory is very efficient, resulting in good performance compared to the empty handler.



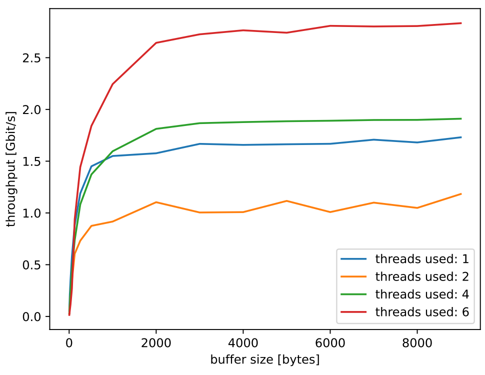
(a) Throughput



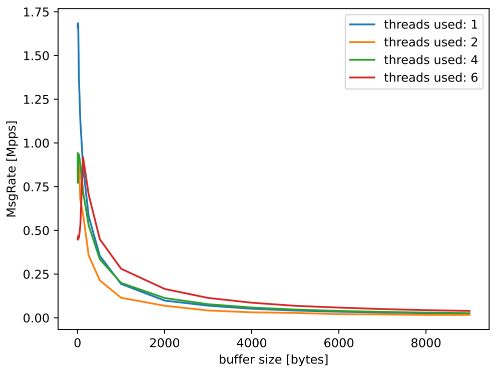
(b) Message rate

Figure 5.10: Throughput and message rate of the aggregate operation using UDP. The maximum theoretical throughput corresponds to the throughput of the RDMA write operation, as it is the last operation of the protocol.

Throughput of Reduce



(a) Throughput



(b) Message rate

Figure 5.11: Throughput and message rate of the reduce operation using UDP

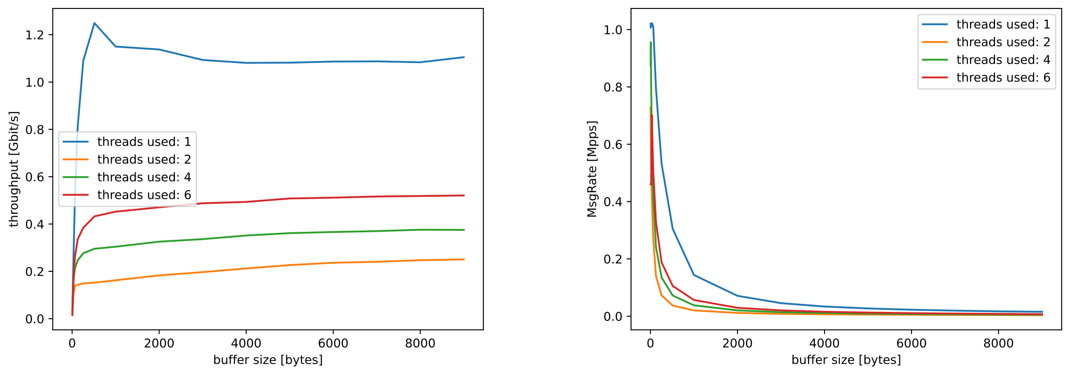
The results of the reduce operation yielded somewhat unexpected outcomes. Compared to the empty handler (Fig. 5.5a) and the aggregate operation (Fig. 5.10a), the throughput significantly decreased. The highest achievable throughput was approximately 2.5 Gbit/s, which is relatively low considering a theoretical maximal achievable throughput of 57 Gbit/s. The primary reason for this considerable performance degradation can be attributed to the atomic additions.

In contrast to the aggregate operation, where the buffer is summed locally, the reduce operation adds each element from the buffer to the shared memory using atomic additions. The applications employed the atomic addition implementation from Caladan [17]. However, these additions appear to be particularly costly on the SmartNIC's DPUs. Further investigation revealed that by replacing the atomic additions with local summing, specifically writing to the thread's local memory, similar performance to the aggregate operation could be observed.

Furthermore, the cycle investigation in Fig. A.1 demonstrates that the throughput is clearly bottlenecked by the reduce operation itself. As a result, the throughput of the reduce operation scales differently than the throughput of aggregate. When utilizing more threads, the throughput immediately increases since employing multiple threads allows to partition the overall computation cost of the reduce operation.

In summary, the additional atomic addition to the shared memory leads to very poor performance.

Throughput of Histogram



(a) Throughput **(b)** Message rate

Figure 5.12: Throughput and message rate of the histogram operation using UDP

In this particular benchmark, the throughput further deteriorated, reaching a mere 1.2 Gbit/s, which is unacceptably low given the current setup. One of the reasons for this decline is the same as in the reduce operation – the use of atomic additions. Employing atomic additions for the histogram operation led to poor performance. However, removing the atomic additions and replacing the increments on shared counters with increments on local counters resulted in improved performance, similar to the reduce operation.

Another bottleneck in this scenario arises from the accesses of random elements of the shared memory. The shared array contains a total of 10'000 counters, with each counter occupying 8 bytes. Consequently, not all counters are allocated on the same 4KB page. Since the buffer sent by the client is randomly initialized before transmission, accessing the counters increases the occurrence of page faults. Additionally, the probability of cache misses also increases. Remarkably, changing the histogram operation to iterate in ascending order over the counters mitigates these issues, resulting in throughput similar to that of the reduce operation.

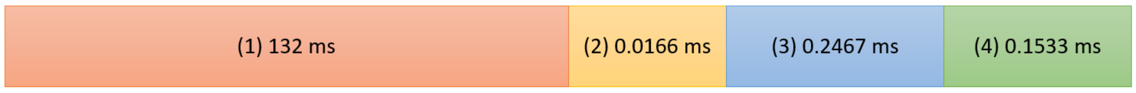
When multiple threads are employed, it's essential to consider that threads share the same virtual address space. Consequently, counter access can interleave and lead to page faults for accesses that would not have triggered a page fault before. The same holds for cache misses. The cycle measurement in Fig. A.1 confirms that the throughput is clearly constrained by the computational cost of the histogram operation itself.

In summary, the atomic additions, along with the additional page faults and cache misses induced by the shared memory accesses of the counters, result in significantly worse performance.

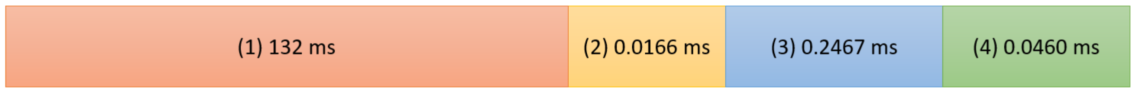
5.4 Critical path break-down of CopyToHost and CopyFromHost in Caladan

In addition, it is worth exploring the breakdown of the process to gain insights into the different stages involved in starting an application in Caladan and identify the most resource-intensive components. The first section in Fig. 5.13 corresponds to the initialization of the process, which involves tasks such as starting the process, reading the config file, configuring settings, and spawning kernel threads. This section is executed only once and is relatively similar for all network processes created with Caladan.

The second part focuses on the scheduler. The measured time represents the average duration between the initialization phase and the execution of the server's main handler function. This part can be invoked multiple times, particularly when working with multiple threads. In the third part, the main server handler creates additional threads and establishes the RDMA connection. Each time a thread is created, the scheduler is invoked, so the measured time in this section includes the scheduler's time.



(a) CopyToHost



(b) CopyFromHost

Figure 5.13: Time spend for the following operations: (1) Initialization of the process, (2) scheduler, (3) setup all threads, (4) CopyToHost/CopyFromHost operation for one packet on the server

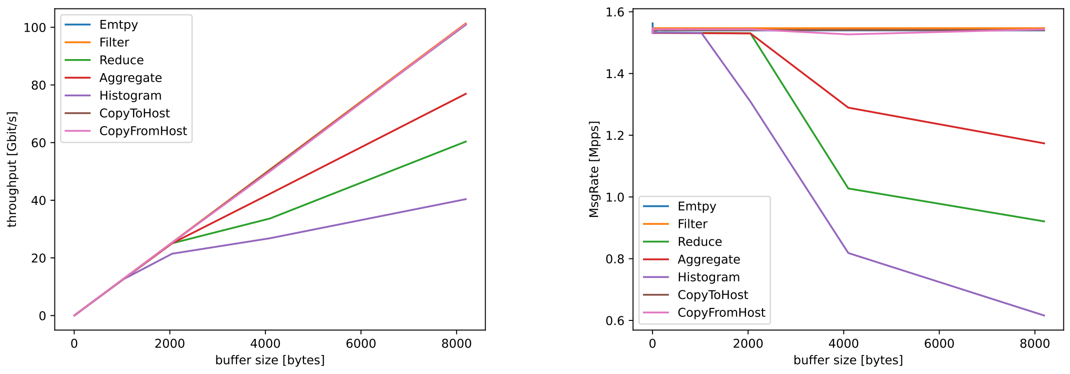
Lastly, the corresponding latency of the operation was measured. It is important to note that this time can vary depending on rescheduling events and the number of packets the server needs to process. However, it is clear that the most time-consuming operation by a significant margin is the initialization process. When continuously switching between applications, this can have a substantial impact on overall performance, as one has to wait a relatively long time before being able to start processing packets.

Processing Unit (PU)	Frequency	ISA	Linux	Caladan	RTOS
Host Ryzen 7 5700	3.8GHz	x86	28576	211	-
BF-2 DPU A72	2.5GHz	ARMv8	13250	192	-
PULP cores (used in PsPIN)	1GHz	RISC-V	-	-	121

Table 5.1: Average latency of context switching between 2 processes. Measurements shown in PU cycles scaled to 1 GHz (i.e., 1 ns/cycle) [26].

5.5 Benchmarks for PsPIN

The PsPIN benchmarks were utilized to measure the following throughputs and latencies. The results aligned with expectations since the examples used for the benchmarks had been previously examined in the PsPIN paper [13]. It is worth noting that the plots for the Filter, CopyToHost, CopyFromHost and Empty are nearly identical and overlap each other. A more comprehensive analysis of the benchmarks can be found in the PsPIN paper [13].



(a) Throughput **(b)** Message rate

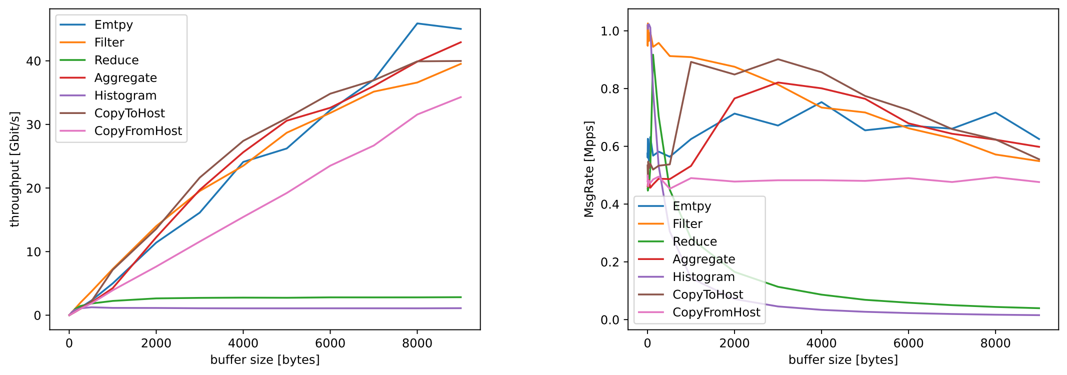
Figure 5.14: Throughput and message rate of PsPIN benchmarks

Chapter 6

Discussion

6.1 Throughput Comparison of Off-path and On-path

Upon comparing Figure 6.1a with Figure 5.14a, a noticeable performance difference between Caladan and PsPIN can be observed. The empty handler, which was optimized to achieve the theoretical maximum throughput of 100 Gbit/s in PsPIN, was unable to reach that level in Caladan. The highest throughput achieved in Caladan was approximately 50 Gbit/s, which is only half of the maximum.



(a) Throughput **(b)** Message rate

Figure 6.1: Best throughputs with associated operation and message rate in Caladan

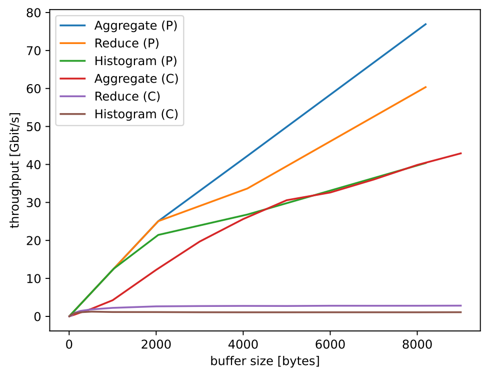
For all IO-bound tasks, PsPIN demonstrates performance nearly equivalent to the empty handler's. Caladan exhibits a similar trend, with almost equal throughput for the filter operation but slightly lower throughput for CopyToHost and also for

CopyFromHost. PsPIN appears to handle RDMA to the Host more efficiently than Caladan.

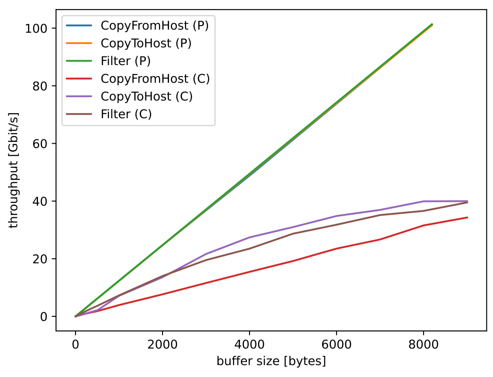
When examining the compute-bound applications, it can also be observed that the results on Caladan and PsPIN follow the same trend, ordered by the increasing complexity of the workloads. In both systems, the aggregate operation performs the best, followed by the reduce operation and the histogram operation, as shown in Fig. 6.2. Caladan manages to keep up with PsPIN for the aggregate operation when comparing the results to the corresponding baseline.

However, this is not the case for the reduce and histogram operations. In these cases, the advantage of the low-cycle accessible memory in PsPIN and its efficient implementation of atomic operations becomes evident. In Caladan, the atomic operation combined with using local memory of the SmartNICs, leads to poorer performance. Nevertheless, it appears that the accesses to the shared memory in random order in the histogram operation also have a significant effect on the performance of PsPIN.

Overall, we can conclude that an off-path system like PsPIN [13] significantly outperforms the on-path offloading utilizing a data plane system like Caladan [18].



(a) Compute-bound applications



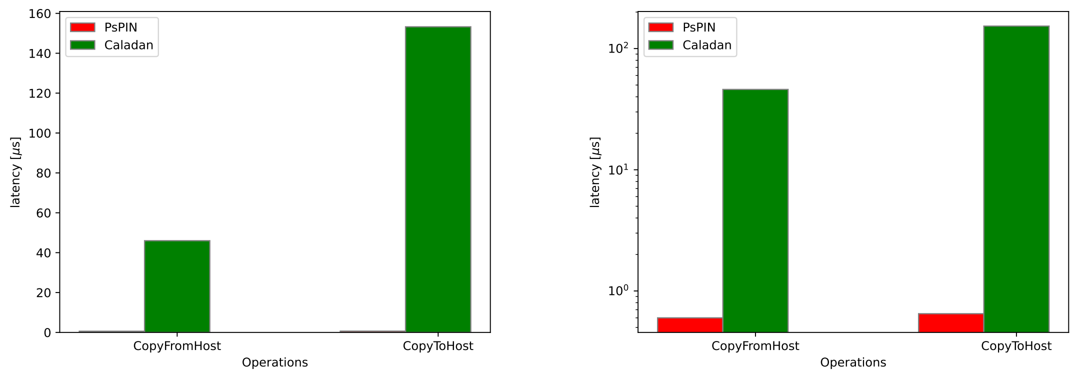
(b) IO-bound applications

Figure 6.2: Performance comparison of compute-bound and IO-bound applications of PsPIN (P) and Caladan (C)

6.2 Latency Comparison of Packet Processing in Off-path and On-path

The disparity in performance between Caladan and PsPIN also becomes evident when examining the latency of the copyFromHost and CopyToHost operations. Figure 6.3 illustrates the time required for a single copyFromHost and CopyToHost operation in the Caladan system, as well as the average feedback delay associated with these operations in PsPIN. As described in the Background section 2.4.1, once the packet handling is complete, a notification is sent back. The feedback time encompasses the duration from sending the packet, processing it, sending the notification, and receiving it. The feedback latency in PsPIN typically falls within the range of 1 μs. On the other hand, Caladan exhibits a feedback latency in the range of 10's μs.

It is important to note that the feedback delay encompasses more than just the latency of a single operation. The discrepancy in latency can be attributed to various factors, including the architectural differences between the SmartNICs. These measurements conclude that PsPIN is more efficient in packet receiving, sending, and performing RDMA operations compared to Caladan.



(a) Without scaling **(b)** Scaled plots using |log(latency)|

Figure 6.3: Latency comparison of CopyToHost and CopyFromHost for Caladan and PsPIN

6.3 Implementation Comparison of PsPIN and Caladan

The significant difference in throughput between Caladan and PsPIN, as observed in the throughput comparison discussed in Section 6.1, can likely be attributed to the varying system architectures and implementations of the off-path and on-path offloading approaches. One notable distinction is the implementation design of the scheduler. The PsPIN scheduler aims to optimize core utilization by incorporating a cluster scheduler and individual schedulers within each cluster to maximize efficiency [13]. In contrast, Caladan's scheduler is specialized for inference but not specifically designed for scaling with network traffic [18]. In Caladan, the number of threads must be defined when starting the process, and the scheduler does not assess whether it would be more efficient to use additional threads. This results in unnecessary overhead when using too many threads, as observed in the measurements. This limitation becomes problematic, particularly when the exact packet quantity is unknown.

Another factor to consider is the difference in header handling and packet receiving. PsPIN requires the necessary information only in the first packet, while Caladan necessitates decapsulation for every packet, requiring the execution of the header handler for each one. Moreover, in Caladan, packet receiving and handling may not occur on the same core, as it employs separate threads called `kthreads` to execute the header handler. This introduces additional latency in the communication between `kthreads` and handler threads [13, 17].

Lastly, it is essential to acknowledge the impact of the ported version and simulation in our comparison. We are evaluating a ported version against a simulation, which means that the ported version may not be fully optimized for ARM CPUs and may still require adjustments to achieve optimal efficiency on the Bluefield DPU. Additionally, the simulation does not perfectly replicate the Bluefield setup, so differences stemming from the setup itself should be taken into account.

In conclusion, these factors, including system architecture, implementation design, header handling, and potential bias due to the ported version and simulation, collectively contribute to the observed performance disparities between PsPIN and Caladan.

6.4 Trade-offs in Off-Path and On-Path Offloading

The comparison between PsPIN and Caladan reveals the trade-offs between performance and application development complexity. PsPIN demonstrates excellent performance, often maximizing throughput in certain scenarios. However, programming applications in PsPIN requires the use of a more complex and limited PsPIN-specific API. This complexity extends beyond just programming, as the entire system setup becomes more intricate due to the absence of a user-friendly operating system.

On the other hand, Caladan's performance falls short with a peak bandwidth of 50% of the maximum possible throughput. However, developing applications in Caladan is comparatively simpler, as it allows for the utilization of all the functionalities and libraries provided by the operating system and also provides a conventional socket API, which makes developing network applications much easier. Additionally, adapting the data plane to specific use cases is more straightforward in Caladan since Caladan is a software and everything resides in user space.

Chapter 7

Conclusion

In summary, we conducted a comparative analysis of on-path and off-path SmartNIC architectures. To achieve this, we first ported Caladan, including the IO-bounded and compute-bounded applications, to the Bluefield-2 off-path SmartNIC. Subsequently, we measured the throughput and latency of these ported applications. In a second step, we fine-tuned the PsPIN simulation model to match our Caladan setup and performed the benchmarks for the same applications tested in Caladan.

Based on the benchmark results, we conclude that PsPIN delivers better throughput performance compared to Caladan. However, it should be noted that achieving this performance with PsPIN requires dealing with increased programming complexity and a more intricate setup. Caladan, while not reaching the same level of performance as PsPIN, offers a simpler development process and the advantage of leveraging the full functionalities of an operating system. Therefore the choice between off-path and on-path offloading depends on the specific requirements and priorities of the application under development. For complex applications that rely heavily on external libraries and additional functionalities, opting for off-path offloading may be the more convenient approach. On the other hand, for relatively simpler applications such as reduce, histogram, or aggregate operations, choosing on-path offloading is likely the better option. These operations can be implemented without significant difficulty, making on-path offloading a viable choice.

Ultimately, the decision between off-path and on-path offloading depends on the complexity and specific requirements of the application. Complex applications benefit from the capabilities of off-path offloading, while simpler operations can be efficiently handled through on-path offloading.

Appendix A

A.1 RDMA reade/write Bandwidth Measurement

Buffer Size [bytes]	NIC-To-NIC [Gbit/s]	NIC-To-Host [Gbit/s]
2	0.03504	0.03536
4	0.07424	0.07328
8	0.14632	0.14816
16	0.2856	0.29976
32	0.58552	0.59808
64	1.2056	1.19856
128	2.29696	2.21472
256	4.52576	4.73576
512	9.41432	9.08072
1024	18.21352	18.4892
2048	37.19384	36.06792
4096	76.35928	57.14
8192	92.798	57.32984

Table A.1: Throughput of the RDMA write operation

Buffer Size [bytes]	NIC-From-NIC [Gbit/s]	NIC-From-Host [Gbit/s]
2	0.0332	0.03256
4	0.06664	0.06632
8	0.14264	0.13896
16	0.29616	0.29408
32	0.55472	0.58312
64	1.16032	1.18552
128	2.29616	2.28168
256	4.58344	4.532
512	8.96512	9.20896
1024	17.21032	18.0248
2048	33.4448	35.1548
4096	74.54784	43.15288
8192	77.26736	47.01256

Table A.2: Throughput of the RDMA read operation

A.2 Cycles Measurements of Caladan Operations

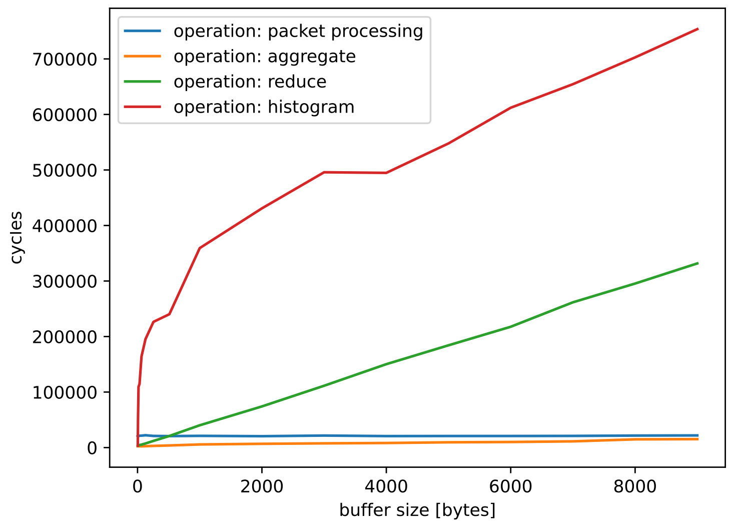


Figure A.1: Cycles of different operations with UDP in Caladan

Bibliography

[1] gettimeofday(2) — linux manual page.

[2] ioctl(2) — linux manual page.

[3] zlib1g Library. Software.

[4] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery.

[5] BROADCOM. Product brief: Stingray ps225. `https://docs.broadcom.com/doc/PS225-PB`, 2022.

[6] Shichao Chen, Christina Delimitrou, and José F. Martínez. PARTIES: QoS-aware resource partitioning for multiple interactive services. In *ASPLOS*, 2019.

[7] Cisco. The New Need for Speed in the Datacenter Network. `http://www.cisco.com/c/dam/en/us/products/collateral/switches/nexus-9000-series-switches/white-paper-c11-734328.pdfdf`, 2015.

[8] Cisco. Cisco Global Cloud Index: Forecast and Methodology, 2015-2020. `http://www.cisco.com/c/dam/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.pdf`, 2016.

[9] Roman Dementiev, Thomas Willhalm, Otto Bruggeman, Patrick Fay, Patrick Ungerer, Austen Ott, Patrick Lu, James Harris, Phil Kerly, Patrick Konsor, Andrey Semin, Michael Kanaly, Ryan Brazones, Rahul Shah, and Jacob Dobkins. Pcm. `https://github.com/intel/pcm/tree/master`, 2022.

[10] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[11] Arm Developer. Arm armv8-a architecture registers: Pmccntr_el0 - performance monitors cycle count register, 2021.

[12] Salvatore Di Girolamo, Andreas Kurth, Alexandru Calotoiu, Thomas Benz, Timo Schneider, Jakub Beranek, Luca Benini, and Torsten Hoefler. Pspin. `https://github.com/spcl/pspin`, 2021.

[13] Salvatore Di Girolamo, Andreas Kurth, Alexandru Calotoiu, Thomas Benz, Timo Schneider, Jakub Beranek, Luca Benini, and Torsten Hoefler. A risc-v in-network accelerator for flexible high-performance low-power packet processing. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[14] DPDK Community. *DPDK: The Data Plane Development Kit*, 2023.

[15] Ethernet Alliance. 2015 Ethernet Roadmap, 2015.

[16] Daniel Firestone. Hardware-accelerated networks at scale in the cloud. `https://conferences.sigcomm.org/sigcomm/2017/files/program-kbnets/keynote-2.pdf`, 2017.

[17] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan. `https://github.com/shenango/caladan`, 2020.

[18] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 281–297, 2020.

[19] Torsten Hoefler and Roberto Belli. Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. Association for Computing Machinery.

[20] Torsten Hoefler, Salvatore Di Girolamo, Konstantin Taranov, Ryan E Grant, and Ron Brightwell. spin: High-performance streaming processing in the network. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, 2017.

[21] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, July 2016.

[22] Intel Corporation. *Intel® 64 and IA-32 Architectures Developer's Manual: Vol. 2B*, 2016.

[23] Internet Engineering Task Force. Computing the Internet Checksum. Technical report, September 1988. RFC 1071.

[24] Jerin Jacob. Github - jerinjacobk/armv8_pmu_cycle_counter_el0.

[25] Bob Jenkins. Hash functions for hash table lookup. *Dr. Dobb's Journal*, 1997.

[26] Mikhail Khalilov, Marcin Chrapek, Siyuan Shen, Alessandro Vezzu, Thomas Benz, Salvatore Di Girolamol, Timo Schneider, Daniele Di Sensi, Luca Benini, and Torsten Hoefler. Osmosis: Enabling high-performance, multi-tenant smartnics in datacenter systems. 2023.

[27] Doug Ledford, Robert Sharp, Hal Rosenstock, Sasha Khapyorsky, Shahar Frank, Michael S. Tsirkin, Sean Hefty, Dotan Barak, Roland Dreier, and Eli Cohen. Rdma-core. `https://github.com/linux-rdma/rdma-core/tree/7a4f9ad9a1d906c5f2bbb18b588309c3d12460ac`, 2020.

[28] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 318–333. 2019.

[29] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-efficient microservices on smartnic-accelerated servers. In *USENIX annual technical conference*, pages 363–378, 2019.

[30] MARVELL. Marvell liquidio iii. `https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-liquidio-III-solutions-brief.pdf`, 2022.

[31] Daniel Molka, Daniel Hackenberg, and Robert Schöne. Main memory and cache performance of Intel Sandy Bridge and AMD Bulldozer. In *Proceedings of the Workshop on Memory Systems Performance and Correctness (MSPC '14)*, pages 4:1–4:10, New York, NY, USA, 2014. ACM.

[32] Gordon E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006.

[33] NVIDIA. Bluefield-2 DPU. `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf`.

[34] NVIDIA. Bluefield-3 DPU. `https://resources.nvidia.com/en-us-accelerated-networking-resource-library/datasheet-nvidia-bluefield?lx=LbHvpR&topic=networking-cloud`.

[35] NVIDIA. FlexIO SKD Programming Guide. `https://docs.nvidia.com/doca/archive/doca-v1.5.0/flexio-sdk-programming-guide/index.html`.

[36] Arjun Ousterhout, Jonathan Fried, Justin Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *NSDI*, 2019.

[37] Georgios Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving low tail latency for microsecond-scale networked tasks. In *SOSP*, 2017.

[38] Z.G. Prodanoff and R. King. Crc32 based signature generation methods for url routing. In *IEEE SoutheastCon, 2004. Proceedings.*, pages 153–158, 2004.

[39] Haoyu Qin, Qiang Li, Jake Speiser, Peter Kraft, and John K. Ousterhout. Arachne: Core-aware thread management. In *OSDI*, 2018.

[40] Luigi Rizzo. netmap: A novel framework for fast packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 101–112, Boston, MA, June 2012. USENIX Association.

[41] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. Fast packet processing with eBPF and XDP: Concepts, code, challenges and applications. *ACM Computing Surveys*, June 2019. 35 pages.

[42] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. Characterizing off-path smartnic for accelerating distributed systems, 2023.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Title of work:

Performance Trade-offs Analysis of SmartNIC Architectures

Thesis type and date:

Bachelor Thesis, September 01, 2023

Supervision:

Prof. Dr. T. Hoefler, M. Khalilov

Student:

Name:	Alessandro Vezzù
E-mail:	avezzu@ethz.ch
Legi-Nr.:	20-914-586

Statement regarding plagiarism:

By signing this statement, I affirm that I have read and signed the Declaration of Originality, independently produced this paper, and adhered to the general practice of source citation in this subject-area.

Declaration of Originality:

`http://www.ethz.ch/faculty/exams/plagiarism/confirmation_en.pdf`

Zurich, 29. 8. 2023: [illegible]

BSc thesis: Performance trade-offs analysis of SmartNIC architectures

Student name: Alessandro Vezzu, avezzu@student.ethz.ch
Advisors: Mikhail Khalilov, E71.2, mikhail.khalilov@inf.ethz.ch
Professor: Prof. Dr. T. Hoefler
Start time: 01.03.2023
End time: 01.09.2023

The final report is to be submitted electronically. All copies and contributed software remain property of the Scalable Parallel Computing Laboratory. If the thesis contains performance experiments, those should follow scientific benchmarking rules. Note that the latency between submitting your thesis (which includes defending it) and grading is up to three weeks.

1. Introduction

In-network packet processing with SmartNICs is an emerging approach to speed-up distributed applications through packet-level parallelism. It follows the in-network computing trend to integrate energy-efficient processing cores with high-speed Network Interface Card (NIC). With such specialized hardware, data processing is partially relegated to the networking infrastructure from the Host CPU. This helps to reduce application latency and maximize Host throughput by hiding computation cost with communication, i.e., maximizing degree of computation/communication overlap.

During the past decade many SmartNIC architectures have evolved along with APIs for their programming. From an architectural perspective they could be classified between two types:

a) **Off-path smartNICs** have an on-NIC PCIe switch that routes packets to multi-core SoC running an OS (e.g., Linux). Off-path SmartNICs offer wider deployment capabilities since applications could use conventional APIs provided by Linux and other libraries. Yet off-path offloading penalizes application latency and throughput due to using routing done through on-NIC PCIe fabric and usage of Linux kernel infrastructure.

b) **On-path smartNICs** feature specialized programmable cores on the packet send/receive critical path path (i.e. NIC link layer). These cores should be programmed with a vendor-specific API. Thus, potential performance gains of on-path offloading compared to off-path one, come with the need to tailor applications for particular offloading API.

Thus, off-path and on-path techniques differ in ease of offloading deployment and degree of possible performance overheads. In this light, the main result of this thesis will be a systematic experimental study of on-path and off-path offloading, that gives a SmartNIC programmer an insight on the best offloading architecture for his application.

2. Project description

The goal of this project is to understand performance and programming/deployment productivity trade-offs that arise from on-path and off-path sNIC architectures.

In particular, 3 experimental setups are considered for performance evaluation:

a) Off-path offloading with Bluefield-3 ARM SoC and Caladan framework.
Bluefield 3 data processing unit (DPU), is the latest generation of Nvidia SmartNICs. It presents the 3rd generation of Nvidia's off-path offloading engine based on ARM SoC that runs Linux. Benchmarking of DPU performance will be done on top of the Caladan library, a load-balancing middleware and between CPU and NIC that also supports generic interface for Remote Procedure Calls offloading: https://github.com/shenango/caladan.
A part of the thesis will be devoted to porting it to the Bluefield ARM SoC.

b) On-path offloading with Bluefield-3 Datatpath Accelerator (DPA) and FlexIO API.
Latest generation of Bluefield DPU also features an on-path Data Processor Accelerator based on RISC-V cores. It could be programmed with FlexIO SDK that is tightly integrated with the RDMA Verbs interface: https://docs.nvidia.com/doca/sdk/flexio-sdk-programming-guide/index.html.

c) On-path offloading with PsPIN.
PsPIN is an open-source smartNIC architecture based on energy-efficient RISC-V cores and features support for streaming processing in the network (sPIN) programming model: https://github.com/spcl/pspin.
sPIN programming model facilitates network offloading through writing packet handlers, a small C-program executed on packets.

The focus of evaluation will be put on answering the following question:
"*What are the fundamental characteristics of workloads that make them better suited for on-path or off-path SmartNIC offloading?*"

To achieve this with Bluefield-based setups (e.g., a) and b)), a set of synthetic micro-benchmarks will be developed for Caladan and FlexIO APIs. Benchmarking suite will cover various performance aspects of a smartNIC and re-implement functionality of PsPIN examples (https://github.com/spcl/pspin/tree/master/examples):

- ability to sustain packet processing for various workloads with various distribution of service times, packet sizes, etc.
- ability to offload IO-bound (e.g. copy_to_host, copy_from_host, filtering) and compute-bound processing (e.g. reduce, aggregate, histogram) workloads;
- RDMA networking primitives (e.g. copy_from_host with RDMA write/read);
- sNIC–Host communication (e.g. host_direct).

For the PsPIN evaluation (e.g., setup c)), microbenchmarks available in the PsPIN Github repository will be re-used.

3. Milestones / project plan

The project plan is divided on 4 main parts:

1. (1.5 months) Off-path offloading benchmarking:
    a. Toolchain setup on Bluefield DPU.
    b. Porting of Caladan library to run on Bluefield-2 DPU ARM cores;
    c. Development of microbenchmarks for Caladan;
    d. Report with performance evaluation of Caladan performance.
2. (1 month) In-path offloading benchmarking:
    a. Setup toolchain for in-path offloading, i.e., Bluefield-3 DPA and FlexIO;
    b. Evaluation of PsPIN using built-in benchmark suite;
    c. Development of microbenchmarks for FlexIO API;
    d. Report on evaluation of DPA performance and comparison to off-path offloading performance with Caladan.
3. (0.5 month) First thesis draft.
    a. Finish the first draft with all the text and experiments by the end of the semester.
4. (remaining time) Thesis submission.
    a. Final revision of the draft text.
    b. Slides for thesis presentation.
    c. Code for open-source publishing.

4. Project administration

Weekly Report: The student is advised, but not required, to write a weekly report at the end of each week and to send it to his advisors. The idea of the weekly report is to briefly summarize the work, progress and any findings made during the week, to plan the actions for the next week, and to bring up open questions and points. The weekly report is also an important means for the student to get a goal-oriented attitude to work.

Final Report: The final report has to be presented at the end of the project and a digital copy needs to be handed in and remain property of the SPCL. Note that this task description is part of your report and has to be attached to your final report. Note that we will need up to two weeks after the presentation to submit the final grade.

sPIN meeting: sPIN weekly meetings are held weekly (Thursdays, 16:30). The student is asked to present the status of the thesis once every month (scheduling is dynamic). The presentation should take 15/20 minutes and leave room for questions/comments.

5. Deliverables

Software:

1. Caladan library ported for Bluefield-3 SmartNIC and documentation for its deployment on Bluefield DPUs.
2. Micro-benchmarks for Caladan evaluation ready for open-sourcing and manuals for running it;
3. Micro-benchmarks for FlexIO evaluation ready for open-sourcing and manuals for running it.

Reports:

1. Final report with systematic and exhaustive performance evaluation of on-path and off-path SmartNICs architectures.

6. Success criteria

1. Caladan library is successfully ported to the Bluefield-2 SmartNIC and the corresponding code is ready for open-sourcing.
2. Performance evaluation of the all Caladan runtime components on the SmartNIC is completed using the benchmarks ported from the PsPIN repository.
3. Detailed experimental and qualitative comparison of the off-path (Caladan running on top of the Bluefield-2 SmartNIC) and on-path (PsPIN simulated with Verilator) architectures is done, resulting in a formulation of application developer guidelines.

[illegible]

Alessandro Vezzu