Linköping University | Department of Computer and Information Science

Master thesis, 30 ECTS | Datateknik

2019 | LIU-IDA/LITH-EX-A--19/092--SE

Implementing and Comparing Static and Machine-Learning Scheduling Approaches using DPDK on an Integrated CPU/GPU

Implementering och jämförelse utav statisk- och maskininlärnings-metod för schedulering med hjälp av DPDK på en integrerad CPU / GPU

Markus Johansson, marjo688
Oscar Pap, oscpa453

Supervisor : August Ernstsson
Examiner : Christoph Kessler

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

As 5G is getting closer to being commercially available, base stations processing this traffic must be improved to be able to handle the increase in traffic and demand for lower latencies. By utilizing the hardware smarter, the processing of data can be accelerated in, for example, the forwarding plane where baseband and encryption are common tasks. With this in mind, systems with integrated GPUs becomes interesting for their additional processing power and lack of need for PCIe buses.

This thesis aims to implement the DPDK framework on the Nvidia Jetson Xavier system and investigate if a scheduler based on the theoretical properties of each platform is better than a self-exploring machine learning scheduler based on packet latency and throughput, and how they stand against a simple round-robin scheduler. It will also examine if it is more beneficial to have a more flexible scheduler with more overhead than a more static scheduler with less overhead.

The conclusion drawn from this is that there are a number of challenges for processing and scheduling on an integrated system. Effective batch aggregation during low traffic rates and how different processes affect each other became the main challenges.

Preface

This thesis was written by Markus Johansson and Oscar Pap and while most of it was co-authored, some of the sections we want to credit individually.

The background about Baseband Device Library, Section 2.1.6, and error correction coding, Section 2.3, was written by Oscar Pap. Section regarding the implementation of the GPU worker, Section 4.2.4, and baseband driver, Section 4.3, was also written by Oscar. Furthermore, the analysis and discussion about the low input test, Section 6.1.1, standard test, Section 6.1.4, and the baseband driver, Section 6.2.1, was also written by Oscar.

Markus Johansson wrote the background about the Cryptography Device Library, Section 2.1.5, Encryption, Section 2.2, and Shared Memory, Section 2.7.3. Markus also wrote about the implementation of the CPU worker, Section 4.2.3, and cryptography driver, Section 4.4. Finally, the analysis and discussion about the medium input test, Section 6.1.2, the stress test, Section 6.1.3, and the cryptography driver, Section 6.2.1, was also written by Markus.

Acknowledgments

We would like to thank our supervisor at Ericsson, Stefan Sundkvist for his assistance during our thesis. We would also like to thank our supervisor August Ernstsson at Linköping University and our examiner Christoph Kessler at Linköping University. Furthermore, we would want to thank our opponents Ammar Alderhally and Martin Jonsson Sjödin.

Contents

List of Figures

List of Tables

Terms and Abbreviations

AES-CTR	Advanced Encryption Standard - Counter Mode.
ARM	A CPU architecture designed for power efficiency.
CPU	Central Processing Unit.
C1	Configuration 1, referring to one of two underlying configurations which all schedulers are tested upon. C1 is less flexible but keeps a smaller overhead.
C2	Configuration 2, referring to one of two underlying configurations which all schedulers are tested upon. C2 is more flexible at the cost of a larger overhead.
DPDK	Data Plane Development Kit.
EAL	Environment Abstraction Layer.
GPU	Graphical Processing Unit.
Jetson Xavier	Reference to the integrated CPU/GPU hardware Nvidia Jetson AGX Xavier on which the experiments of this thesis are tested on.
LDPC	Low-Density Parity-Check.
ML	Machine learning, and will also be used as reference to the scheduler implementing machine learning.
Round-robin	Reference to the simple baseline scheduler implementing a round-robin distribution.
RX	Input buffer/queue.
Static	Reference to the scheduler which distribution algorithm is based in observed and theoretically expected behaviour, and are static compared to the ML algorithm.
TX	Output buffer/queue.

1 Introduction

With the fifth-generation wireless telecommunications technology (5G) on the way, the data rates and latency of the telecommunication wireless network will see major improvements [1]. These ambitions mean that the amount of data packets forwarded throughout the network needs to be handled and processed at a higher rate. Such developments require new and innovative ways to improve and utilize both hardware and software in radio access network (RAN) stations.

By smart utilization of hardware in the forwarding plane, the processing can be accelerated for heavy and large amount of computations. With this in mind, the computation power of GPUs is of interest to accelerate the packet processing in the forwarding plane for common tasks such as encryption and baseband. As power efficiency and processing speed increases in combination with the size of these platforms decreasing, opportunities to implement these in network forwarding stations becomes feasible [2]. A major contributor to these improvements is the integrated CPU/GPU platforms. At the time of writing, one such recent release is the Nvidia Jetson Xavier [3].

With the addition of hardware, portability also becomes an important aspect. For scalability, the implementation should be able to utilize and work on any underlying hardware of a RAN station. To support this, the DPDK[1] framework - initially an Intel developed collection of libraries but now open-source - will be used to implement the abstraction and interface for the hardware platforms, which the software can communicate through. DPDK also provides many libraries and infrastructure for packet processing, such as buffers and packet representation.

Furthermore, the addition of multiple platforms with distinct characteristics results in a need for smart and efficient scheduling of processing tasks across the heterogeneous systems. In order to evaluate this, a scheduler will be implemented on top of the Xavier hardware platform. By considering properties such as packet size, latency and packet rate the scheduler can then dispatch the packets to the GPU, or process them directly on the CPU cores.

In order to conclude important decision points and factors to consider, a static ration distribution algorithm will be evaluated. This static ratios scheduler will, in turn, be compared to a machine learning algorithm, which will explore the available platforms and scheduling possibilities using reinforcement learning. It is also of interest to analyze the use of DPDK as our underlying software

[1] Data Plane Development Kit

and the impact of flexibility in terms of overhead. Finally, to provide insight in the scheduling effect on power efficiency the consumption is measured across scheduler implementations utilizing the hardware resources differently.

1.1 Motivation

Mobile networks are used all over the world and are the cornerstone for the networked society, where everything is moving towards connectivity. To support the vast amount and diversity of data expected in future networks, Ericsson, which is one of the world's largest manufacturers of equipment for building networks for mobile communications, is developing products to drive and support the networked society. The subjects of this thesis are defined to investigate and develop algorithms, architecture, tools, etc. to support a huge increase in speed and IoT data for Radio Access Networks.

In order to provide an interface for communicating and dispatching tasks, the Data Plane Development Kit (DPDK) will be used. DPDK is a framework that can be used to speed-up packet processing in e.g. a telecommunication network [2] [4]. Previous research [5] has shown that, compared to a native Linux stack, DPDK can reduce the latency with up to 90%. The study compared the two different techniques on a UDP-based game server where low latency is crucial for a good gaming experience.

DPDK also suits the context of supporting multiple hardware platforms, since the framework contains abstractions that are portable to any hardware. With the many hardware alternatives becoming feasible options for radio network stations, portable solutions that work for varying hardware constructions is desirable. [2]

There is a current hype of machine learning and the possible benefits to be reaped in various fields and tasks. Scheduling is one such area, where reinforcement learning is often evaluated to explore and produce adaptive scheduling decisions [6, 7]. While the state space and actions of a scheduler can be very complex, the ambition is to capture the important aspects in a relatively intuitive and straight-forward approach utilizing the DPDK context. On top of the performance results and feasibility, the machine learning algorithm compared with the theoretical approach can provide insights in evaluating the theoretical implementation.

Power efficiency is also an important factor which holds great weight in modern ambitions of not only improvements in speed but also environmental footprint. Together with the cost and logistic concerns regarding often remote and constantly active RANs, the scheduling effect on power consumption should be considered.

The scheduler will be constructed in C, in order for it to communicate with DPDK, which is also implemented in C.

1.2 Aim

The purpose of this thesis is to implement a DPDK abstraction on top of our heterogeneous system consisting of a GPU and a CPU. Since encryption and baseband are both tasks desired to be applied to packets in the 5G processing pipeline, these will correspond to the work performed by the application. While DPDK does provide libraries, called poll mode drivers (PMD), they lack the desired algorithms AES-CTR for the GPU and LDPC for the GPU and CPU. These will, therefore, have to be constructed by us and then integrated into the DPDK ecosystem through their API. The application's performance will then be investigated with regards to throughput and latency. This is done on the NVIDIA Jetson AGX Xavier which is an embedded system-on-module computer. The thesis will also implement and compare two different scheduling implementations. One of the implementations

[2] `https://www.dpdk.org/about/`

will be based on the theoretical properties of the different platforms available and the other will utilize a machine learning algorithm. A comparison of different DPDK configurations will also be made using these schedulers to evaluate the trade-off surrounding a more dynamic task scheduling with the cost of extra overhead in the DPDK context. To provide some final insight in the utilization of resources in terms of power efficiency, power consumption measurements are gathered for each of the schedulers during comparable conditions.

1.3 Research Questions

This thesis aims to answer these questions:

1. Given data packets of varying sizes and demands, how well does a static DPDK scheduler implementation on top of a heterogenous system perform based on the theoretical properties (such as speed, throughput and power efficiency) of each platform compared to a simple round-robin baseline?
2. In a DPDK implementation on top of a heterogenous system, how well can a self-exploring adaptive machine learning scheduler perform compared to a simple round-robin baseline?
3. In a DPDK-based implementation, does a flexible scheduler with more overhead benefit compared to a more static scheduler with less overhead?
4. For comparable throughput conditions, how is power consumption affected by the utilization of the different resources provided by the Jetson Xavier?

1.4 Delimitations

For the scope of the thesis, the major limitation was time, and this did, therefore, produce a number of limitations on where development focus is placed. While the concept of a scheduler and its potential is quite broad, below is a list of limitations that we consider relevant to further simulate and evaluate a realistic scheduler:

1. Processing drivers are not state of the art, resulting in a somewhat skewed relation between optimal platform performance.
2. Processing tasks is limited to AES-CTR encryption and LDPC decoding.
3. The target platform is limited to the Jetson Xavier integrated CPU/GPU.

1.5 Thesis Overview

The layout of this thesis is as follows: In Chapter 2 the background and theory is presented. Chapter 2 also includes the terms and technologies important for this thesis. Related work for this thesis is presented in Chapter 3. In Chapter 4 it is explained thoroughly how DPDK was implemented on the hardware and how the schedulers were created and implemented. In Chapter 5, it is explained how the implementations were evaluated and in which environment they were tested. Chapter 6 includes the results of the evaluations, which are also analyzed and discussed. The final chapter, Chapter 7, is where the conclusion and future work is presented.

2 Background

This chapter will in more detail introduce and explain terms, software, hardware, and techniques needed to understand the work in this thesis. Related work that is relevant to this thesis is also presented and discussed.

2.1 Data Plane Development Kit

Data Plane Development Kit (DPDK) is a set of libraries that was created in 2010 by Intel and was later made open-source in 2013 under the Linux Foundation, more information about this can be found on the project website [1].

A mobile network generally consists of three planes (parts): the data plane, the control plane, and the management plane. The data plane carries the network traffic i.e. the payload. The control plane is responsible for routing and defines what to do with the payload. The management plane carries the administration traffic that is needed for network management [8].

DPDK is used in the data plane and its job is to accelerate the process of deciding what to do with arriving data and also performing some tasks on the data itself. DPDK consists of libraries and drivers that can be used for accelerating packet processing on all major CPU architectures, making it easily portable. One of the important components of DPDK is the Environment Abstraction Layer (EAL). EAL hides the different hardware specifics and gives the programmer a generic interface to work with. The EAL is then responsible for all the accesses to the available hardware, thread management, and memory allocation. This is what makes it possible to port DPDK to different hardware environments. More information about how DPDK works can be found on the project website [2].

A Previous study [9] has shown that the DPDK library can be used to create software routers where a throughput up to 40 Gbit/s are possible for packets with a size of 1024 bytes, compared to a throughput of 6.3 Gbit/s for a software router created using the Linux kernel stack. It is possible that the DPDK router would have shown even greater throughput if not the maximum capacity for the Network Interface Card (NIC) would have been 40 Gbit/s, so the router could not get data faster than that [9].

[1] `https://www.dpdk.org/about/`
[2] `http://doc.dpdk.org/guides/prog_guide/overview.html`

2.1.1 Network Packet Buffer Management

Whenever a data packet is picked up by a DPDK application through the RX (receiving) port, a `rte_mbuf struct` is allocated from a designated memory pool [3]. These mbuf structs are what will represent the packet during its lifetime in the DPDK application and are provided by the Mbuf library, which is part of DPDK. It is accessed by passing around a pointer to the `rte_mbuf` struct it resides in, avoiding any unnecessary copying of data. The struct itself contains metadata information such as data length and sequence number and is in memory followed by the packet data, possibly with some headroom in between. The Mbuf library also provides a bunch of macros and functions for accessing and modifying the packet data, such as pointers to the beginning of data showcased in Figure 2.1. An application can then drop or forward the packet on some TX (transmitting) port, which in both cases result in freeing the pointer and returning the `rte_mbuf` to its pool of origin for future use.

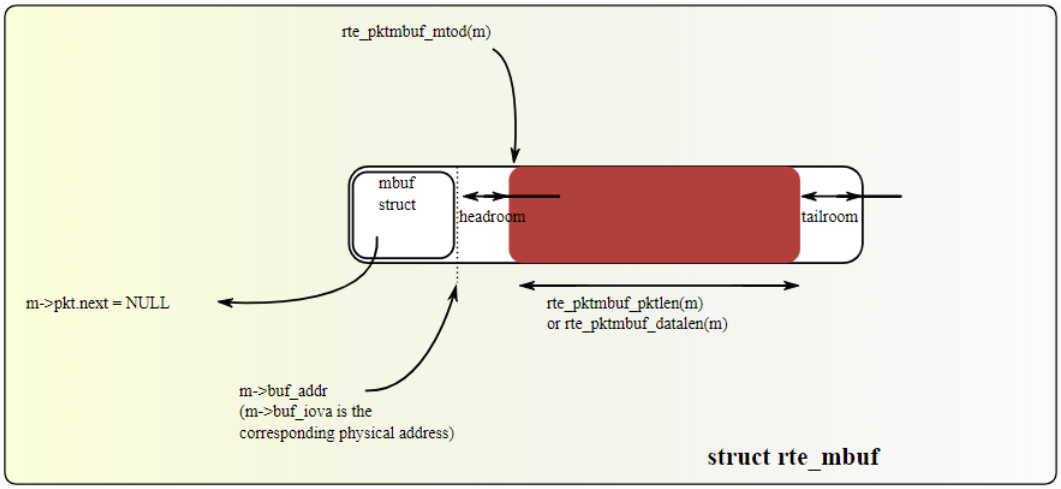


Figure 2.1: Message buffer structure. Image downloaded from `https://doc.dpdk.org/guides/prog_guide/mbuf_lib.html` in June 2019.

2.1.2 Memory Pool Manager

For performance reasons, DPDK utilizes compile-time allocated memory pools through the Mempool library. This means DPDK provided data objects, such as a `rte_mbuf`, which lifetimes are limited to the DPDK application, are not dynamically allocated during run-time. The Mempool library also helps the programmer with certain memory optimizations such as padding to ensure that allocated objects are spread equally among channels and ranks in memory[4]. The memory pools can also be initiated with a local cache for even faster memory access. In Figure 2.2 follows a short code example showcasing the creation of a `rte_mbuf` memory pool and allocation of objects from the pool during run-time.

2.1.3 Ring Buffer

To manage the storage of objects in parallel environments, the DPDK provides ring buffers. The DPDK Ring library allows the management of queues through ring buffers [5]. The ring buffer is a circular buffer that is lock-free, thread-safe and uses the first in, first out method to manage the buffer. A circular buffer is a fixed-size array where the last element in the array is linked to the first element of the array, thereby creating a circle. The buffer uses two pointers, one head pointer, and

[3] `http://doc.dpdk.org/guides/prog_guide/overview.html`
[4] `https://doc.dpdk.org/guides/prog_guide/mempool_lib.html`
[5] `https://doc.dpdk.org/guides/prog_guide/ring_lib.html`

```c
#include <rte_mempool.h>
#include <rte_mbuf.h>

/* create the mbuf pool,
*  pool_size and cache_size refers to the number of rte_mbuf
*  objects the pool will hold
*/
struct rte_mempool *pktmbuf_pool =
    rte_pktmbuf_pool_create("pool name", pool_size,
    cache_size, sizeof(struct rte_mbuf),
    RTE_MBUF_DEFAULT_BUF_SIZE, rte_socket_id());

struct rte_mbuf *local_buffer[n];

/* allocate n rte_mbufs from pktmbuf_pool and store their
*  pointers in local_buffer.
*  ret corresponds to rte_mbufs actually retrieved,
*  which could differ from n if there are fewer than n free
*  rte_mbufs inside the pool for example
*/
int ret = rte_pktmbuf_alloc_bulk(pktmbuf_pool, local_buffer, n);
```

Figure 2.2: Memory Pool Code Snippet

one tail pointer, to write and read from the buffer. When data is written, the head pointer is moved up and when data is read, the tail pointer is moved up. The reads and writes can be performed by multiple cores at the same time in a multi-producer/multi-consumer fashion and the Compare And Swap (CAS) technique is used to ensure this is done correctly. On top of application-specific usage, these rings are also used throughout the DPDK libraries.

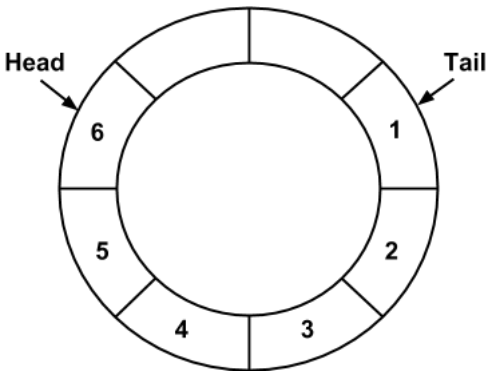


Figure 2.3: Ring buffer

2.1.4 Poll Mode Driver

In a DPDK application, a Poll Mode Driver (PMD) is responsible for receiving, processing and transmitting the packets in the application [6]. The PMDs come with an API that is used to configure the available hardware devices and their corresponding queues. When a packet is received in the RX port, the PMD retrieves the packet, processes it with some action, and then transmits it forward

[6] http://doc.dpdk.org/guides/prog_guide/overview.html

through the TX port. The PMD library also serves as an integration API of external resources for packet modification. The cryptography device library and baseband device library, which is explained more thoroughly later in this chapter, are both PMDs.

PMDs are often used because they are latency-efficient. This is due to constant polling for data to process which removes the overhead of interrupts. The drawback of this method is that it uses more CPU resources. Another aspect of PMDs is that they run in user-space instead of kernel-space, which means that it bypasses the Linux kernel networking stack and its associated overhead, which can be a bottleneck, and communicates directly to the network hardware instead. Therefore, the network devices must be unbound from the kernel and instead bound to a DPDK driver.

2.1.5 Cryptography Device Library

The cryptography device library (Cryptodev) provides a framework for managing and supporting hardware and software with cryptography drivers and also defines generic APIs that support different cryptographic operations such as cipher and authentication [7]. Cryptodev was one of the first device libraries to be implemented into DPDK in 2015. Today, Cryptodev has support for 18 different drivers that utilize different cryptography techniques and has support for different hardware, but most of the drivers are created for an Intel x86_64 CPU.

2.1.6 BaseBand Device Library

BaseBand Device Library (BBdev) is a framework for wireless workloads [8]. BBdev was developed to create a generic acceleration abstraction framework that supports both hardware and software with acceleration functions. BBdev was added to DPDK in 2017, the same year that DPDK became open source under the Linux Foundation. Since BBdev is newer than Cryptodev, the number of supported drivers are not as developed compared to Cryptodev. At the time of writing this thesis, there were only two drivers available, a null driver and a software turbo driver. The null driver is a minimalistic implementation of a software BBdev driver and does not modify the data in any way. The software turbo driver, created for Intel CPUs, utilizes the turbo technique for workload acceleration and support different encoding and decoding operations, such as Cyclic Redundancy Check (CRC). CRC is used for error-detecting which means that if the payload data was changed in any way during the transmission between the nodes, CRC can detect it and can, within some limits, correct the data to its original value. This is explained further in Section 2.3. Error correction can accelerate the data transmission since the receiving node does not need to request the incorrect data again, but can instead correct it automatically itself.

2.2 Encryption

Encryption is the process of encoding a message so that only the intended recipient can read and access it. Ensuring data protection and secure communication between devices is one of the key aspects of the 5G system [10]. There are multiple encryption algorithms that will be included in the standardization of 5G, for example, AES-CTR, SNOW 3G and ZUC, which are also used in the 4G system and are well-proven to be secure.

2.2.1 Advanced Encryption Standard

Advanced Encryption Standard (AES) is a cryptographic algorithm [11]. It is approved by the Federal Information Processing Standards and is widely used today. AES is a symmetric block cipher which means that the data is always encrypted in blocks with the same size and that the same key is used

[7] https://doc.dpdk.org/guides/prog_guide/cryptodev_lib.html
[8] https://doc.dpdk.org/guides/prog_guide/bbdev.html

for both encryption and decryption. For AES, this means that the data is always encrypted in blocks of 128 bits, but the keys that are used to encrypt and decrypt can either be 128, 192 or 256 bits [11].

AES can be executed with different block cipher modes for encryption. Examples of the most common ones are Electronic Codebook (ECB), Cipher Block Chaining (CBC) and Counter (CTR).

AES-CTR is parallelizable by nature and multiple papers examine different ways in how to get further speed-up compared to the original version [12, 13].

The counter mode uses a random initialization vector called counter vector (CTR) which is the same length as the block size. The CTR is then XOR:ed with a counter i which creates the keystream $K_i = CTR \oplus i$. This key stream is encrypted with AES ($E_K(K_i)$) and XORed with the message m_i. This equals that the cipher block $c_i = m_i \oplus E_K(K_i)$.

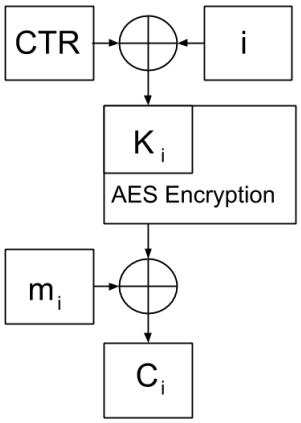


Figure 2.4: AES-CTR encryption

2.3 Error Correction Coding

Error correction coding is how errors that have been introduced in data during the transmission through a communication channel can be corrected when the data is received [14]. During the twenty-first century, the field has been revolutionized with methods capable of approaching the theoretical limits of performance: the channel capacity. Today, every packet transmitted over the internet is error correction encoded which the receiver can use to determine if an error of some sort has been introduced into the data during the transmission. [14]

Examples of error correction codes that have been accepted in the 5G standardization are LDPC, turbo code, and polar code [15]. In the 3G and 4G networks, turbo codes have been the primary method for error correction but to be able to handle the high throughput requirements for 5G, LDPC codes will be more commonly used [16]. Turbo codes and LDPC codes are similar to each other but the computation of LDPC can be divided into a large number of independent operations. It is, therefore, better for parallelism and for increasing throughput [16]. In a survey by Shao et al. [17] they conclude that polar code does not achieve as high throughput as LDPC unless you tamper with the code correction capability.

2.3.1 Shannon Limit

The Shannon limit is an upper limit for how many bits of data can be transmitted without error per second over a channel with a given bandwidth when the signal is exposed to an uncorrelated noise [18]. The channel capacity can be calculated with Shannon´s formula:

$$C = W * \log_2(1 + p/N) \text{ bits/second} \tag{2.1}$$

Where W is the bandwidth in Hz, P is the average signal power in watt and N is the power of the noise [18]. When choosing an error correction method for when transmitting data over a transmission channel, it is desired that the method has a transfer rate as close as possible to the Shannon Limit.

2.3.2 Low-Density Parity-Check

Low-Density Parity-Check (LDPC) is an error-correction method [19]. With the help of LDPC, the receiver of the message can understand if the data has been changed in any way and even calculate which bits were changed and what their actual value should be. LDPC code consists of the data to be sent and *parity* bits. A parity bit is a bit added to make sure the number of ones in the data is either even or odd, depending on preset preferences. Most data bits are connected to multiple parity bits and when a parity check fails, the information from the parity bits can help to retrieve the original data bit [19]. Data transmissions utilizing LDPC has been reported to be close to the Shannon limit [20] and should, therefore, be a suitable choice as a method for transmitting data over some transmission channel.

Jung et al. present a high throughput process and architecture for LDPC code encoding in their paper, suited for the IEEE 802.11 (WLAN) standards [21]. In their paper, they use a block length of 1944 bits with which they measured 7.7 Gbps throughput.

In a paper by Wang et al. [22], an LDPC code decoder is presented which has a throughput of 490 Mbps when block lengths of 1944 bits are used. In their work, they use an iterative algorithm and this speed was obtainable when doing 5 iterations. More iterations give better error correction but lower throughput. The number of iterations should, therefore, be set depending on how much noise there is. Decoding LDPC code is very computationally intensive and because of its parallelism characteristics, it is better suited to run on a GPU than on a CPU [22].

2.4 Scheduling

Scheduling is the process of assigning tasks to different resources. A scheduler is responsible for the activity of scheduling tasks to the workers and keeping track of the goals defined by the scheduling algorithm. Schedulers can have multiple goals, such as throughput and latency, but since these goals often contradict each other, there have to be some compromises made. A simple scheduling algorithm is the first in, first out (FIFO) algorithm which performs no rearranging of packets [23]. When a packet is received, it is queued and the packet first in the queue is the one to be processed next. Another basic scheduling algorithm is round-robin. It is used for simple load balancing work between available workers [24]. It provides a more dynamic solution than FIFO since objects processed are no longer necessarily bound by other objects in the queue.

2.4.1 Latency

Latency is an important aspect of all data traffic where some applications benefit from it by just running more smoothly at lower latencies, others require it for any useful functionality. Packets that are delivered too late might be considered dropped by the receiver and be requested to be sent again. This downgrades the efficiency of a cell tower since it has to send the same package

twice even though it was not dropped, just late. Any scheduling algorithm which aims to simulate a real-world implementation must, therefore, consider latency and its correlation with throughput. Especially for a heterogeneous system, such a property becomes a very distinct decision point, where the nature of the system and the varied latencies of the traffic creates high impact choices. Through knowledge of the system's properties, latency-aware processing can be maintained while still utilizing other benefits of the system such as accelerated throughput.

2.4.2 Reorder

Data traffic can require or benefit from having packets arrive in an expected order. If the packets are sent out of order, it can affect the TCP protocol which will think that packets were lost and have to be re-sent again, which downgrades the latency and throughput. Packets can end up out of order when they are processed by a multi-core system, where the processing of packets is done asynchronously. A valid decision point for the scheduler can therefore be if a packet needs reordering since not all packets necessarily need reordering and depending on the traffic rate of packets requiring order. By distributing in-order packets efficiently, the reordering impact on overall throughput and individual packet latency can be improved, while order misses can greatly punish performance in unnecessary long wait times for processed data.

2.5 Packet Classification

While simple first-in, first-out approaches were used by most network routers for a long time, modern day packet forwarding has higher demands on the quality of service (QoS) [25]. Services such as routing therefore classify data packets upon entering their input buffers.

Packet classification corresponds to the mapping of data packets to certain rules. These rules, in turn, determine things such as priority and destination of the packet, basically distributing the packets in different flows. The mapping can, in turn, be made based on a number of factors such as packet size, QoS requirements or other meta information [25].

2.6 Packet Distribution

Network packets can have a different amount of data as payload. To understand what sizes of data are most common, we can look at two real-world examples that are being updated regularly. Seattle Internet Exchange (SIX), which is a neutral and independent internet exchange point, shows statistics about the distribution between the different packet sizes [26]. At the time of when this thesis was written (spring 2019), the chart looked like shown in Figure 2.5.

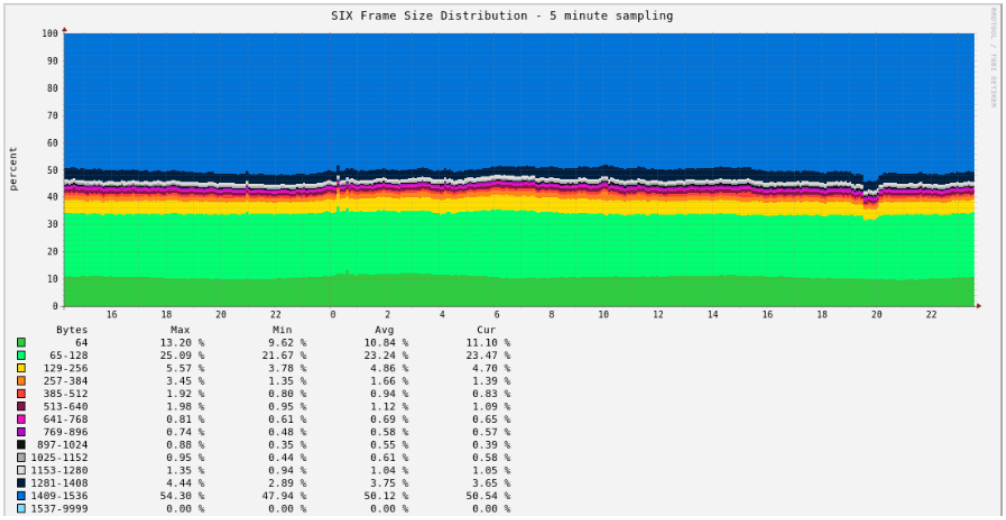


Figure 2.5: Seattle packet distribution

This can be compared to how the packet sizes are reported at the Amsterdam internet exchange point at the same time [27], which is shown in Figure 2.6.

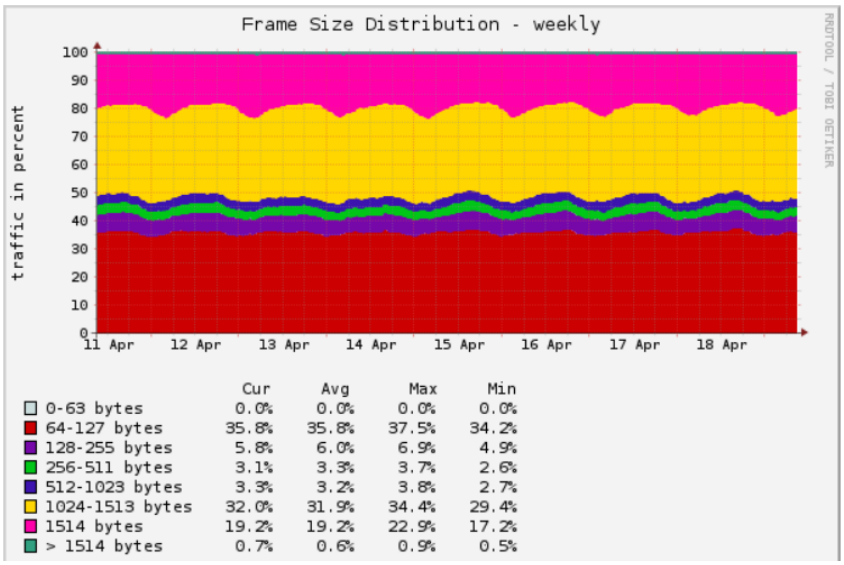


Figure 2.6: Amsterdam packet distribution

Packets of smaller sizes, around 64 bytes, are generally TCP control packets that are used to establish a connection, acknowledge a message or tell the receiver that there are no more data to be received. The bigger packets, around 1514 bytes, are data transferring packets. When data is being transferred, it is desirable to send as much data as possible with every packet. Therefore, it is no surprise that the most common packet sizes are the smaller control packets or the bigger data transferring packets.

2.7 GPU Hardware

As the power efficiency and general-purpose applicability increases, the Graphical Processing Unit (GPU) is becoming an important and promising component for performing heavy tasks in short

periods of time. The strength of the GPU does however come with different approaches, compared to modern CPU's, to consider in terms of memory architecture and data configurations.

2.7.1 Parallel utilization of GPUs

The concept of parallelism is about being able to execute multiple instructions simultaneously [28].

Modern GPUs often have thousands of threads among hundreds of cores they can use to process data with while the CPU is more limited with usually only a handful of physical cores. However, while the GPU has many more threads to use, all threads are usually bound to execute the same instruction, but on different data (SIMD). The CPU has fewer threads to utilize but can instead execute multiple and completely different instructions on different data (MIMD).

As mentioned by Maghazeh et al. [29], there are limited benefits of parallel computing when processing a single packet. It is therefore important that the GPU get enough packets so that it can utilize a sufficient number of warps[9] for decent occupancy. The GPU needs bigger batches of data packets compared to the CPU, which in its sequential nature does not require any batch aggregation.

Packet processing has the characteristic of being highly parallelizable at the packet level and memory intensive, which means that memory communication latency between CPU and GPU is critical [30]. Tseng et al. [30] show that a GPU can give 40 times better throughput than a single CPU core when it comes to heavy computation tasks, in this case, 4 hash computations. Hash computations were chosen as a method because it is a common task to do when processing software network packets [30] .

2.7.2 Jetson Xavier

Section 1 briefly presented the Jetson Xavier, which is the system and basis for the study. The Xavier is an integrated CPU/GPU consisting of an Nvidia 512-core Volta GPU [10] and an 8-core ARM v8 64-bit CPU. For more detailed information, please refer to cited documentation sources.

2.7.3 Shared Memory

Unlike the integrated CPU-GPU hardware Jetson Xavier, a heterogeneous system deploying a CPU and a GPU would also maintain physically separate memory spaces. Communication between the devices in such systems is made through the low-bandwidth PCI-Express bus, making copying of memory back and forth costly with relatively high latencies. The acceleration gains provided by the GPU computations must therefore outweigh the expensive data transfers.

With the shared DRAM of single-chip integrated CPU-GPU systems, the CPU and GPU can access the same memory, completely avoiding the costly PCI-Express bus path. This makes GPU acceleration of tasks with hard real-time constraints more feasible. However, with the shared memory come other challenges. Since the hardware now accesses the same memory space, high contention might occur if both CPU and GPU perform memory intensive work. This can, in turn, reduce the performance of the application significantly [31].

2.7.3.1 Tegra

The Jetson Xavier uses the Tegra memory architecture, shown in Figure 2.7.

[9] A set of GPU threads which do the same thing at the same time on the same SM, SIMD

[10] `https://developer.nvidia.com/embedded/dlc/jetson-agx-xavier-thermal-design-guide`

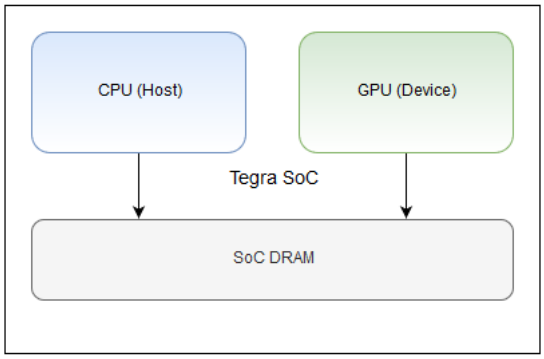


Figure 2.7: Tegra memory architecture

When allocating data for an NVIDIA GPU, the programmer has the option to do so utilizing four different memory types. These are listed in Table 2.1 together with behavior corresponding to the hardware part of the Tegra architecture in the Jetson Xavier.

Table 2.1: Tegra memory type behaviour

Memory Type	CPU	GPU
Device Memory	Not directly accessible	Cached
Pageable Host Memory	Cached	Not directly accessible
Pinned Host Memory	Cached	Uncached
Unified Memory	Cached	Cached

With these properties listed, the programmer should select the proper memory type when allocating data in order to fully utilize the integrated system. This means device memory for any data buffers which are only used by the GPU, and vice versa pageable memory for CPU. Pinned and unified memory is not as straightforward, but the caching of unified memory comes with overhead in terms of coherence and other cache maintenance operations. So for any buffers which are either small, only accessed once on the GPU or whose access patterns are not cache friendly, the pinned host memory is preferable [11].

2.8 Machine Learning

Machine Learning is seen as a subset of artificial intelligence (AI) and is the study of algorithms and statistical models which computer systems can use to perform a specific task without using explicit instructions. A machine learning algorithm uses available training data to make predictions and decisions without being told exactly how to do the task.

There are many different types of machine learning methods which have different advantages and disadvantages and are suited for different types of environments. One of these methods is the reinforcement learning method.

2.8.1 Reinforcement Learning

Reinforcement Learning is an area of machine learning where a software agent will try to learn the cause and effect from different available actions and try to maximize the reward from them [32].

[11] Tegra Memory https://docs.nvidia.com/cuda/cuda-for-tegra-appnote/index.html#memory-management

Normally, machine learning algorithms have a dataset for training from which it detects patterns for the optimal actions for each state. Reinforcement learning does not use a training dataset, instead, it learns by trial and error. In a more unpredictable environment where right or wrong is harder to define, reinforcement learning should, therefore, be a well suited method of choice [33].

The agent is not told what actions to take but must learn itself which actions bring the most reward by trying them out itself. The actions taken might not give the biggest reward right now, but can affect future actions and future rewards to get a total higher reward over an extended time. One of the bigger challenges with reinforcement learning is the trade-off between exploration and exploitation. The agent must use what it already knows to maximize the reward, but it must also explore new actions in order to find actions that give better rewards [32].

2.8.2 Markov Decision Process

One process for how an agent will decide what the best action is by using a Markov Decision Process (MDP) [34]. The MDP is suited for problems where the agent can make a decision given only the current state. This means that the agents' next state is independent from what has happened in the past, given the current state. In the context of packet scheduling, this approach is suitable since each packet can be considered as a distinct action. While one could consider a continuous state space during the entire application, it is hard to define. There is also not necessarily any clear gain in keeping complex relations between packets, as long as the state space can represent the overall load and status of the resources.

An MDP consists of a five-element tuple $(S, A_s, P_a, R_a, \gamma)$:

- $/S/$ - a set of states
- A_s - a set of actions available for state s
- $P_a(s, s')$ - the probability that state s will lead to state s' given action a
- $R_a(s, s')$ - the immediate reward given when transitioning from state s to state s' as a result from action a
- $\gamma \in [0, 1)$ is the discount factor which represent the trade-off between now and future.

The goal is that the agent will have a policy for all possible states it can get into and the optimal action for that state. For this to happen, the agent must find the optimal action for each state by trial and error. When the agent finds an action that brings a better reward for a certain state, it updates the policy with that information. More training allows the agent to try more states and actions and increases the possibility that it will find the optimal action for each possible state [34].

3 Related Work

This chapter lists and discusses useful and related works to the subject of this thesis. The related works covers subjects such as data processing on heterogeneous systems, machine learning algorithms for offloading computational work and task scheduling, packet classification on different systems and also packet processing in a system where the CPU and GPU have unified memory.

3.1 Exploiting Integrated GPUs for Network Packet Processing Workloads

Offloading network packet processing from the CPU to the GPU is something that has been researched numerous times. Tseng et al. [35] investigated the possibilities and performance potential for offloading packet processing on an integrated GPU. Integrated GPUs have several advantages over discrete GPUs. Instead of communicating through the PCIe buses, an integrated GPUcommunicates on an integrated communication channel which offers lower latencies than through the PCIe buses. Shared memory between the CPU and the GPU is also one significant advantage since it reduces the number of times data has to be copied back and forth between the platforms. According to Tseng et al. [35] a good packet processing system should be able to handle packets in parallel and have low latency for communication between host and device, demands that are met by an integrated GPU system.

The results of their paper were their framework where at least one CPU core is responsible for network tasks and communicating with the GPU, the rest of the CPU cores and the GPU are responsible for processing the packets. Their experiments showed that the CPU I/O core became the bottleneck. They were able to achieve the highest throughput when three CPU cores fed packets to the GPU. Their framework improved the throughput by 2-2.5x compared to a CPU-only solution. The workload that was evaluated consisted of only lightweight computations and heavier computations might have benefited the GPU system even more [35].

The results from this paper are highly related to our study; that the integrated GPU system was faster than the CPU-only system is an interesting result with regards to our system. It is also interesting that they achieved the highest throughput with three CPU cores feeding packets to the GPU. It might be interesting to test if we can gain a higher throughput with multiple CPU cores handling the I/O traffic and communicating with the GPU, than having those cores processing packets.

3.2 Processing data streams with hard real-time constraints on heterogeneous systems

Verner, Schuster, and Silberstein [36] evaluate a heterogeneous hardware system consisting of a CPU and a GPU. The GPU is used to accelerate the processor, which in this case is AES-CBC encryption, of data streams. They propose an algorithm that they call Rectangle, which heuristically selects a rectangular area of streams in a two-dimensional space based on data processing rate and the deadline of the work. Once a partition has been made, meaning the streams inside the rectangle are sent to the accelerator while the rest is processed by the processor, the framework checks if the subsets are schedulable. A set of jobs are schedulable if there are no streams inside the set that misses their deadline and dependencies are enforced based on the schedule.

If they are schedulable, the partitions are done and the work can begin by aggregating the batch of data that is to be processed by the accelerator and sending it there. Through performance models, the work is load balanced on the GPU in order to optimize throughput of the batch and complete before the set deadline of the batch. This is done by creating warps of similar work size per thread and partitioning the warps so that each Streaming Multiprocessor (SM) has a similar work size and finally partitioning the warps among thread blocks based on work size.

They show that their polynomical-time Rectangle method can provide stable throughput for varying workloads. It was especially efficient for workloads with a large number of streams and short deadlines.

This article provides insight into the possibilities as well as the obstacles of accelerating packet processing by utilizing a GPU. It is closely related to our work, however, our constraints on the processing might differ, and our system is a bit different where the GPU is embedded in the system.

3.3 Machine Learning-Based Runtime Scheduler for Mobile Offloading Framework

Since this thesis aims to investigate machine learning implementations for scheduling, earlier studies researching this topic are important as they provide a starting point. Eom et al. [37] evaluate 19 different machine learning algorithms for offloading computational work to an external server from mobile devices. They consider four parameters as relevant: computation cost, data size, network bandwidth and arguments required for setup (between client and server). The first three of these were then combined into one based on a formula, resulting in two input parameters. In order to simulate a mobile network condition, they also had variations in network bandwidth and latency by setting up the communications across three different setups. These were LAN, campus network and an Amazon EC2 instance.

They gathered 640 data points, where each data point is one execution of the offloading framework. The datasets were then labeled with local or offload, together with the two input parameters. For example, if offloading a 1920×1080 image into a machine with a GPU in LAN takes a shorter time than local processing, the instance is labeled as offload. 70% of these data instances were then used as training, while the rest were used for the test set. They then evaluated the accuracy of each algorithm, trained on the training set, with the test set. With this set, they then evaluate both an offline offloader as well as online.

The result shows the feasibility of implementing machine learning algorithms in the context of scheduling for mobile offloading frameworks. They also conclude that the Instance-Based Learning algorithm performed best for offline offloading. This algorithm was then also used to showcase the potential of online offloading implementation.

The problems are very similar, the main difference being them offloading to an external server. This means they have to take into account communication setup, latency, and bandwidth. Furthermore, their data is classified, while our approach is rather to classify what the right decision is using machine learning. This means we can not directly utilize their data.

3.4 Machine learning based online performance prediction for runtime parallelization and task scheduling.

Li et al. [38] present an adaptive framework called ASpR (Adaptively Scheduled parallel R) that can be used for task scheduling based on past performance data. It works by retraining the performance model every time new execution history is obtained so the model gets more precise the more data it gets. The weights are updated based on the comparison between the predicted and observed values, which results in the error between predicted and observed values being reduced.

They also conclude that loops are more difficult to predict, and therefore their ASpR framework test drives the loop first to predict the execution cost. The test, in turn, is only efficient to use if the loop has enough iterations since the overhead created must be compensated. If the iterations are too few, the loop is just divided equally between the threads.

This paper is interesting for how they approached the problem of how to use Artificial Neural Networks for task scheduling. The study is relevant for us since the goal is similar in that a machine learning approach is utilized to predict the result of scheduling decisions. Their prediction is however not - at least directly - surrounding packet processing, but rather analyzing the code itself.

3.5 Delay-Optimal Computation Task Scheduling for Mobile-Edge Computing Systems

In a paper by Liu et al. [39] they study the optimization problem of mobile-edge computing where an agent must decide if it is most optimal to perform a task locally or offload it to a server for cloud computing. In their paper, the agent must consider queuing state, execution state and the state of the transmission unit before making a decision for how to schedule a task. The goal for the agent was to minimize the delay of each task and power consumption at the mobile device. The result of the paper was that their scheduling algorithm performed better than the greedy scheduling policy where tasks were scheduled to the local processor whenever the processors were idle.

The optimization problem of analyzing where it is optimal to perform a task is the same problem as will be analyzed in this paper. Liu et al. have bigger overheads to consider, but the basic logic is the same and how they used the Markov decision process is an approach that is relevant to this thesis.

3.6 A Reinforcement Learning Strategy for Task Scheduling of WSNs with Mobile Nodes

The scheduling of tasks in a wireless network with mobile nodes is something that Cirstea et al. [6] study in their paper. The problem they study is how each mobile node can learn which task to perform to use the available resources in the most effective way. Each agent has four different actions to choose between with different corresponding rewards for each action.

To solve this problem, Cirstea et al. used the Markov Decision Process based technique Q-learning. This method was chosen because it only needs knowledge of the current state and not previous states to make a decision. Each agent has a Q-table where states, action and corresponding reward are stored. The agent can then use this table as a look-up table to find out which action to take for the current state.

The result of this study was that their algorithm could reduce energy consumption by 60% while still keeping the quality of service at the same level. They also found that their algorithm was capable of adapting quickly when the environment changes, which is important when the agents are mobile and the environment can change quickly.

The problem of task scheduling and using the available resources in an optimal way is highly related to our paper. Cirstea et al. also presents the benefits of using a Markov Decision Process, more specifically Q-learning, and the possible gains from this method, which we will take inspiration from when creating our reinforcement learning method.

3.7 Optimizing Many-field Packet Classification on FPGA, Multi-core CPU, and GPU

Qu et al. [40] presents three decomposition-based implementations for packet classification. Decomposition-based classification means that the header is split up in multiple parts, each part is operated on individually and is then merged with the other parts to create the final result. This means the decomposition technique consists of three phases, preprocessing, searching and merging. The advantage of this technique is that during the searching phase, different algorithms can be explored and different data structures can be used, such as hash tables.

Different techniques for how to achieve high throughput on FPGA-based, multi-core General Purpose Processor-based (GPP) and GPU-accelerator-based classifiers are presented. The platforms use rule sets to classify the packets. The rule sets consists of different rules which tells what actions to do with the packet, such as where to forward it or how to modify it.

The results were that for small rule sets, FPGA is better than GPP and GPU with regards to throughput and latency. This is because for small rule sets the on-chip memory can be used. For larger rule sets, off-chip memory must be used but this results in longer latency and lower throughput. Multi-core GPP is better for large rule sets since it can support it on-chip, the drawback being that when it processes a large batch of packets, the processing latency increases for every single packet. GPU-accelerated packet classifiers are also more suited for bigger batches of packets because of the communication overhead. Processing batches increase the throughput, even if it means that a lot of data is transferred between the host and device memory. The drawback of GPU-accelerated packet classifiers it that it is hard to optimize both the host and kernel functions at the same time.

The article is helpful in understanding the pros and cons of relevant hardware and how they relate to some properties of packets. It indicates that big batches of data should be processed on the GPU to make up for the communication overhead. In our system, the GPU and CPU have shared memory so the communication overhead is much smaller and smaller batches of data is needed to utilize the speed up the GPU can bring.

3.8 Latency-Aware Packet Processing on CPU-GPU Heterogeneous Systems

Maghazeh et al. [29] present a software architecture for CPU-GPU heterogeneous systems for latency-sensitive applications. They used, similar to this thesis, a system where the CPU and the GPU has a unified memory architecture which removes the bottleneck of transferring data through the PCI-e bus. In the paper, Maghazeh et al. raise the problem of packet classification, which is described in Section 2.5, and the problem with having a fixed batch size. A fixed size might lead to higher latency when the input rate is lower than the batch size because the system will then wait until a large enough number of packets has arrived.

The result of this paper is the technique that Maghazeh et al. present [29]. To solve the problems described, they have a CPU thread that monitors the rate of which packets arrive and selects a suitable batch size from this information. They also have a kernel that does not terminate (a persistent

kernel) that adapts itself at run-time and processes the batch of data on the GPU. By using a persistent kernel, they avoid the overhead of launching new GPU kernels every time the batch size is changed. With the help of experiments, Maghazeh et al. showed that their approach reduces the packet latency on average by a factor of 3.5, when compared to more common, fixed-sized batch, solutions. They conclude that their technique is interesting for any applications where the tradeoff between throughput and latency is important and it takes more than one item to utilize the GPU sufficiently.

The results are directly applicable to our thesis, which will also have to consider the trade-off of throughput and latency when aggregating the GPU batches. However, it is unlikely our implementation will go as far in developing this dynamic approach due to our larger system with multiple kernels, but nevertheless, lessons can be drawn.

3.9 Protecting real-time GPU kernels on integrated CPU-GPU SoC platforms

With an embedded CPU/GPU platform it is relevant to consider how the sharing of main memory between CPU applications and GPU kernels affect the performance, which is examined by Ali and Yun [41]. They also present their own framework BWLOCK++ that is designed to reduce the memory bandwidth contention problem in integrated CPU-GPU architectures. To make sure that the GPU kernels performance is not affected by the CPU cores contending memory, their framework throttles the CPU when the CPU runs memory intensive applications. Their work is based on critical tasks being performed by the GPU and it is therefore important that the GPU is prioritized.

The results indicated that the framework could reduce the execution time to approximately a third [41]. This experiment was done on Jetson TX2 which has a memory bandwidth of 58.4 GBps [42].

The paper by Ali and Yun is interesting for understanding how to schedule tasks on our system and what our bottleneck might be. The hardware for this paper is the Jetson Xavier which has more than twice the memory bandwidth with 137 GBps compared to the Jetson TX2 [3]. Therefore the bottleneck might not primarily be the memory bandwidth but it is interesting to take that into consideration.

3.10 Research and Implementation of High Performance Traffic Processing based on Intel DPDK

Zhe et al. [43] present a high performance traffic processing method based on the DPDK platform, referred to as Intel DPDK but is currently open-source and will for consistency with the rest of this study therefore just be called the present name of DPDK. The goal of their method is to improve the traffic processing with the help of the techniques used in DPDK. They state that there are multiple sources of bottlenecks for a network processing system, where the usage of the Linux kernel is one of the main ones. A bottleneck which DPDK does not have because it bypasses the Linux kernel by using its own data plane library to send and receive data packets. They compared their method against a system based on the Linux kernel stack and one system based on the PF-RING platform, which is a high speed packet capture library [43].

Zhe et al. tested and evaluated these three systems based on packet loss rate when tested for a traffic load between 0 and 10 Gbps and also packet sizes between 64 and 1500 bytes. The results of their study were that the system based on the Linux kernel performed worst for both tests, with a loss rate at almost 90% when the traffic rate reached 10 Gbps. The DPDK system performed the best with 0% loss rate up to 8 Gbps and approximately a 5% loss for 10 Gbps as a worst case. DPDK also performed best when tested with different packet lengths and had an approximately 5% loss rate for 64 B packets as a worst case [43].

The results of this paper show the superiority of DPDK in data processing capabilities and performance. The paper also describe the structure of their DPDK system and which key aspects of the DPDK framework they used to utilize the possibilities of a DPDK system and the speedup it brings.

4 Implementation

This chapter introduces the general structure of the system, how the data flows and the decisions about the design and implementation that were made.

4.1 Application Setup

The application is initiated with two sets of command line arguments. The first set corresponds to the DPDK Environment Abstraction Layer (EAL). This configures the application environment, such as how many cores and ports are enabled for use and can also initiate poll mode drivers (PMDs). The second set is application specific. For our scheduler, which performs baseband and encryption processing, this can correspond to algorithms used or other properties such as cipher key length and block sizes.

Once running, the application configures started PMD devices, buffer-queues, ports and tasks for each core. When setup is complete, all configured cores enabled by the application are started through the `rte_launch` API. Each core launches a conditional while-loop which performs tasks corresponding to the configured role. This will be explained further in Sections 4.2.3, 4.2.4, and 4.7.

4.2 Application Structure and Flow

The application maintains three different task loops, each corresponding to a core role. These are the `scheduler_loop`, the `cpu_worker_loop` and the `gpu_worker_loop`. The CPU core assigned to schedule and distribute packets run the `scheduler_loop`. The worker loops run two PMDs each, one baseband driver and one crypto driver. While the packet flow is basically the same for the CPU workers and the GPU worker, there is substantial logic and setting differences to justify separate implementations. For example, the desire of larger packet batch sizes for GPU kernel launches requires larger storage arrays compared to the smaller burst of packets processed by CPU workers.

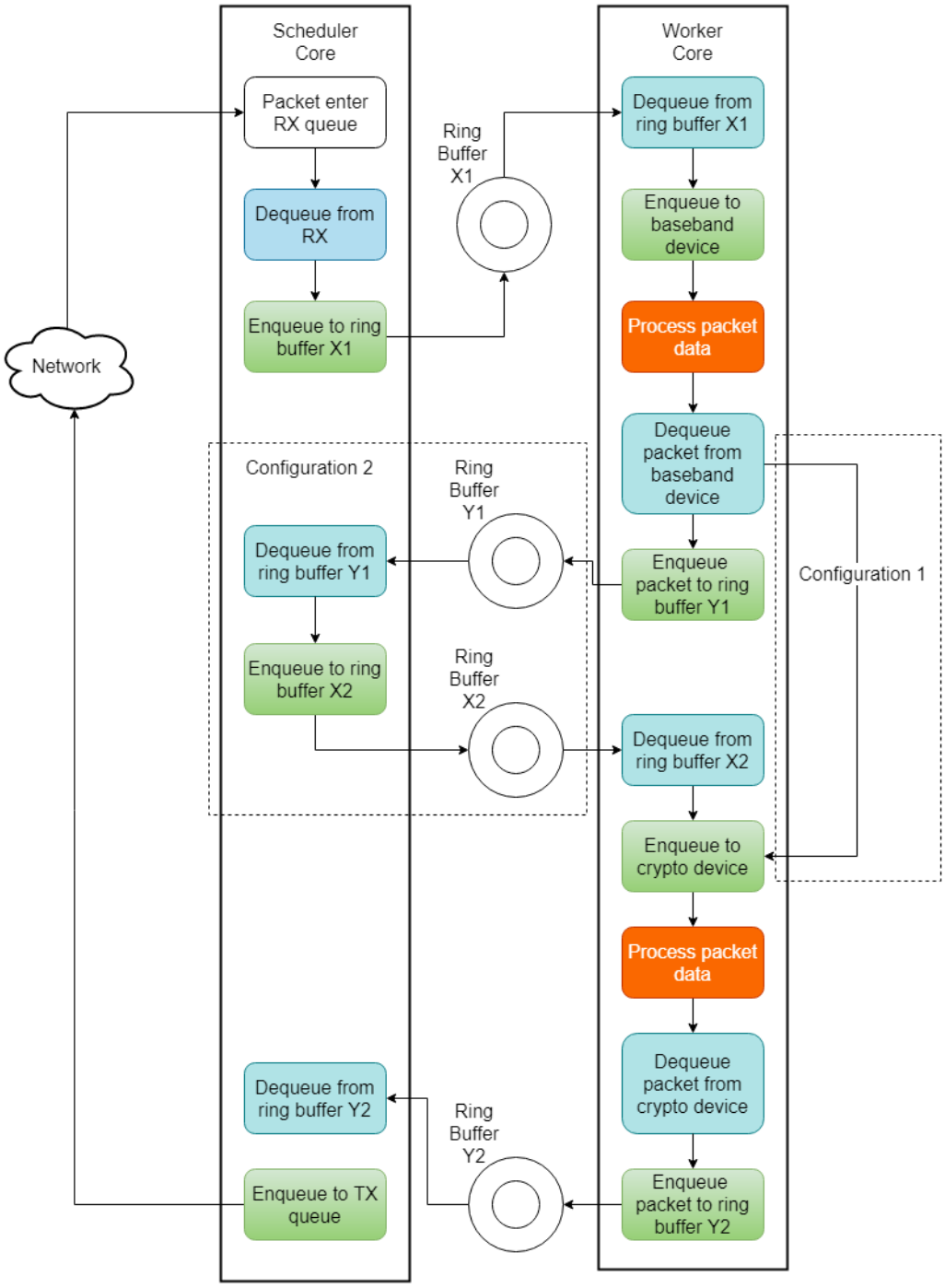


Figure 4.1: Simple packet flow

Leaving out processing details and decision logic of the scheduler, the flow of a packet through the application can be described in a simplified way as shown in Figure 4.1. An important highlight is the areas labeled *Configuration 1* (C1) and *Configuration 2* (C2). These correspond to the differences in DPDK setup, where C1 only schedules **once** while C2 performs **two** scheduling steps. This means C1 has less scheduling overhead but is also not as flexible since the scheduling decision in-

cludes both baseband and encryption combined. Meanwhile, C2 can distribute the packets across platforms based on individual tasks. This also affects the scheduling algorithms, which will have some varying behavior depending on which configuration they run on.

Note that while it is not explicitly stated, each core maintains its own buffer arrays for temporary local storage between dequeues and enqueues, which will be further explained in Section 4.7.

4.2.1 Distribution

For distributing packets among multiple workers, DPDK ring buffers were utilized. The ring library supports non-blocking CAS operations and multi-producer(MP)/multi-consumer(MC) access, making them well suited for data distribution in a multi-core environment.

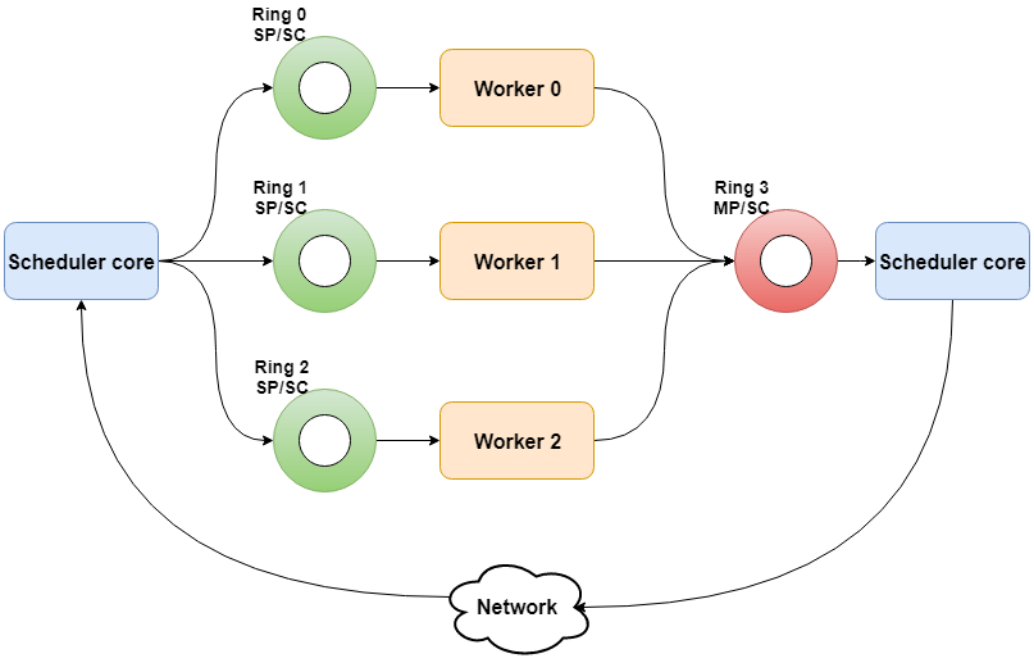


Figure 4.2: Distribution overview

The rings are created during application setup when roles are assigned to each of the enabled cores. Figure 4.2 showcases the setup. Each worker core is assigned an input ring buffer, which the scheduler core distributes packets to. The workers then poll this ring for packets, and upon retrieval process them and enqueue the processed packets on the output ring buffer. All workers produce to this output ring, which the scheduler core polls.

4.2.2 Packet Table

As for any modern service wishing to uphold QoS requirements, the scheduler maintains a table of packet types, mapped to different packet properties. These packet types are later on used to differentiate important characteristics for decision making in scheduling the packets on the CPU or GPU.

Table 4.1: Packet types

Type	Packet size (B)	Reorder	Latency sensitive
1	0–255	Yes	Yes
2	0–255	No	No
3	256–1024	No	No
4	256–1024	Yes	No
5	>1024	No	No

The packet types are selected with the ambition to each identify a distinct property which impacts the scheduling decision. The packets handled by the scheduler will be further discussed later on in this chapter when the test application is presented. The test application is responsible for generating packets and will attempt to do so in order to provide a distribution falling into each of the above packet types.

Sizes are quite straightforward. Varying sizes impact the computing required and therefore the decision. Reordering is a common feature of data handlers which manages order-reliant flows. Latency requirements are also common, for example in TCP traffic. It also simulates a prioritization of packets, similar to how mobile network traffic differentiates VoIP traffic and internet traffic.

All of these properties are, while specifics not explicitly anchored in any specific real-world example, meant to provide general decision points which would be important to a real RAN scheduling traffic.

4.2.3 CPU Worker

With the task of processing smaller bursts, the CPU worker has a pretty straight-forward logic. The individual CPU worker cores do not perform any parallel processing of multiple packets at a time in terms of software multi-threading or hardware-threading, resulting in a sequential packet by packet processing. Any bunch of packets received from the scheduler is directly enqueued on the attached CPU driver, which process the packets. The CPU worker then polls the drivers' output ring for any finished packets and enqueue them on the worker output ring. Each CPU worker is initiated with a designated CPU core on which it runs, and as far as our knowledge goes DPDK places no limit on how many cores can be supported.

4.2.4 GPU Worker

The GPU worker, responsible for receiving packets from the scheduler and enqueue them on the designated GPU driver, runs on a separate CPU core similar to the CPU worker. It also performs the same processing tasks as the CPU workers. However, rather than a sequential packet by packet processing, the GPU implementation performs its work on multiple data packets in parallel. Since populating the GPU and running a kernel comes with a certain overhead compared to processing directly to the CPU, a demand for a sufficient batch size is required to utilize the parallel computation power of the GPU. For the Volta 512-core GPU part of the Jetson Xavier system, a maximum batch size of 128 packets was chosen. This size simply corresponds to the approximate of how many bytes the GPU can process in parallel with the available threads, which is in turn decided by the kernel processing code. Once a desirable size is buffered, or a wait time threshold has been reached, it is sent to the GPU worker which in turn enqueue the data on its designated driver. The processed packets are then polled by the worker in the same manner as described in Section 4.2.3.

The GPU workers created for this thesis utilizes an external library to process packets in some way. The external library is a pre-compiled shared CUDA application, which is linked to the driver. In this context, the driver is simply a middle-man between the encryption library and the scheduling application, utilizing the DPDK driver API. An overview of the structure, without mentioning details of the scheduler application, can be seen in Figure 4.3.

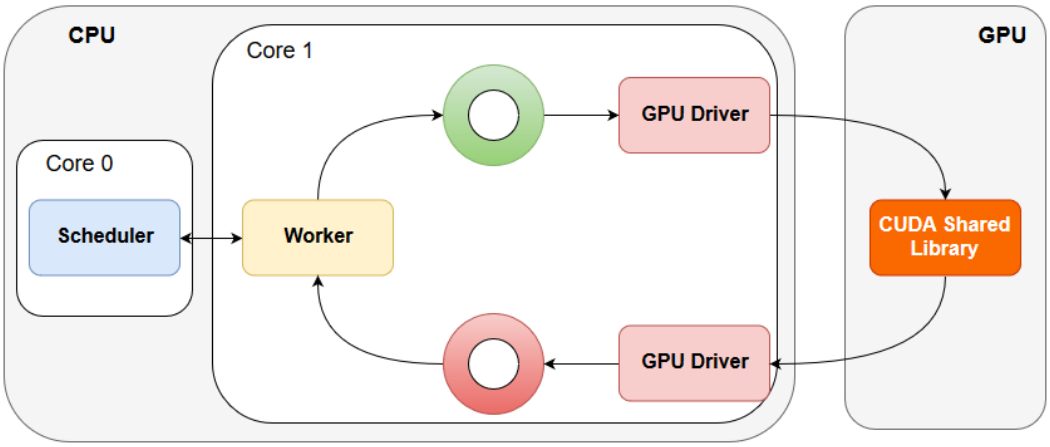


Figure 4.3: GPU driver overview

4.3 Baseband Driver Implementation

The first driver that a data packet encounters is the baseband driver. In this thesis, two baseband drivers have been implemented that decode packets with the LDPC method. This method was chosen because it is a common method that will be used in 5G networks to handle the higher throughput that 5G will result in. It is also better suited for parallelization than what, for example, turbo code is. The big difference between the baseband drivers is that one is adapted to work on the GPU and that the other is made for the CPU cores. The implementation is based on the LDPC decoder presented by Wang et al. [22] and can be found on GitHub[1]. The decoder had to be altered some to fit into the DPDK architecture and our application. Both the CPU and GPU driver utilizes our own developed external libraries and are simply the middlemen that distribute the data to the libraries.

4.3.1 CPU Driver

The LDPC decoder presented by Wang et al. [22] was written for GPU and not CPU, therefore the code had to be re-written to be able to work on only the CPU. The CPU driver runs on the ARM64 architecture in our hardware and utilizes our external LDPC library to decode one packet at the time. Depending on the input rate, the driver can poll up to 32 packets from the ring, but the decoding is done one packet at the time. Decoding LDPC code is heavy computationally and best utilized on a GPU, the CPU was therefore expected to decode the data slower than the GPU.

4.3.2 GPU Driver

The GPU driver also utilizes an external LDPC library which is based on the work presented by Wang et al. [22]. The code had to be adapted to the DPDK framework and parts not needed were removed, but the algorithm was kept consistent. The GPU driver decodes multiple packets at the same time and should, therefore, get a significant speed-up compared to the CPU implementation, where the difference should be biggest for larger packets.

4.3.3 Baseband Driver Measurements

To better understand the baseband drivers thresholds for latency and throughput the CPU and GPU drivers were tested for different packet sizes and also an Internet Mix of packets (IMIX). The IMIX distribution is based on the measured packet distribution presented in Section 2.6, this means that the smaller packets of 243 MB stand for 40% of the incoming data traffic, the medium sized 486 MB

[1] `https://github.com/robertwgh/cuLDPC`

packets stand for 20%, and the big sized packets 1458 MB stands for the rest, 40% of the traffic. The results from these experiments are later used when the theoretical static scheduler algorithms were created.

Table 4.2: Latency for baseband

Platform	Packet size (B)	Average (ms)	Median (ms)	Std dev (ms)
CPU	243	105.6	105.5	0.45
	486	211.0	210.6	0.96
	1458	630.0	630.0	0.58
	IMIX	337.1	336.9	0.69
GPU	243	9.4	9.4	0.19
	486	12.5	12.6	0.22
	1458	17.8	17.6	0.41
	IMIX	12.3	12.2	0.26

As can be seen in Table 4.2, the GPU performs approximately 11 to 35 times lower latency than what the CPU does on average. The IMIX results are most noteworthy because these tests are most similar to reality.

Table 4.3: Throughput for baseband

Platform	Packet size (B)	Average (Mbps)	Median (Mbps)	Std dev (Mbps)
CPU	243	4.7	4.7	0.0
	486	4.7	4.7	0.0
	1458	4.7	4.7	0.0
	IMIX	4.7	4.7	0.0
GPU	243	53.0	52.9	1.0
	486	79.6	79.4	1.4
	1458	168.6	169.8	3.8
	IMIX	130.0	131.0	2.6

The results in Table 4.3 shows that the GPU performs 11 to 35 times higher throughput than the CPU. The CPU performs the same throughput for all different packet sizes because each byte takes the same time to decode, therefore, a packet of size 486 bytes takes twice the time to decode than a packet of size 243 bytes. This is not the case for the GPU, which gets a significant speed-up for bigger packets.

The CUDA implementation was also analyzed with the help of CUDA tool nvprof, which is a profiling tool that allows you to collect data from the CUDA kernels. To better utilize the GPU, it is important to achieve a high level of occupancy, which is the ratio of the number of active warps per processor to the maximum number of possible active warps. According to the CUDA Best Practice Guide [2], a low occupancy leads to higher memory latency and results in a worse performance than what is possible with high occupancy. In the guide, they also state that a higher occupancy does not always lead to better performance, which means that it is not necessary to achieve a 100% occupancy to fully utilize the possible GPU speed-up. The LDPC CUDA implementation measured a minimum of 67%, a maximum of 81% and a 74% average of occupancy and while there is no goal target, this indicates that the implementation utilizes the GPU in a satisfactory way.

[2] `https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html`

4.4 Cryptography Driver Implementation

With the ambition of utilizing the parallel nature of the GPU, the two cryptographic drivers implement AES-CTR. On top of that, due to the opportunity for parallel acceleration, it is also relevant for 5G networks to fulfill the hard QoS requirements. Since no AES-CTR drivers existed for either GPU or ARM64 CPU, these drivers had to be developed by us. The actual encryption source code is based on an open-source project[3]. This code was in turn integrated into the DPDK driver API.

4.4.1 CPU Driver

The CPU driver, which runs on ARM64 architecture in our hardware, takes a batch of packets and performs the AES-CTR encryption sequentially on one packet at the time directly inside the driver. This is partly for convenience, but since the source code for the CPU version of the AES-CTR encryption were written in C, there was also no reason to separate driver and library. Since processing is done directly on the CPU it is fast and effective at processing small packets, where the computation is not too heavy. However, the distribution of packets between scheduler, CPU worker and CPU driver is, if the packet rate allows it, done in batches of maximum 32 packets. This is mainly for optimizing the overhead around moving packets back and forth from the ring buffers, which requires an atomic modification of head and tail pointers. By enqueuing and dequeuing in batches, this modification can be done only once for multiple packets, rather than once for each.

4.4.2 GPU Driver

In contrast to the CPU driver, which ran its processing code directly inside the driver, the GPU driver utilizes an external library. On top of the external library, the only substantial difference is the batch aggregation. While the CPU driver transferred packets in bursts up to 32 packets, the GPU tries to aggregate up to 128 packets. The aggregation of batches and transfer of data between CPU and GPU is further explained in Sections 4.5 and 4.6.

4.4.3 Cryptography Driver Measurements

In order to gain an understanding of drivers performance and their behavior on each platform, tests were made. This data is also used to support chosen thresholds and decisions attached to the scheduler algorithms.

Table 4.4: Latency for crypto

Platform	Packet size (B)	Average (ms)	Median (ms)	Std dev (ms)
CPU	243	1.73	1.73	0.000
	486	3.3	3.32	0.000
	1458	9.8	9.81	0.005
	IMIX	5.26	5.32	0.170
GPU	243	1.77	1.77	0.021
	486	1.99	1.97	0.056
	1458	3.26	3.18	0.140
	IMIX	2.29	2.29	0.005

Table 4.4 shows that the CPU has a slightly lower latency for 243 byte packets but significantly higher latency for the other packets compared to the GPU. Standard deviations show no notable anomalies, but overall the CPU has basically no deviations at all while the GPU maintains a slight deviation of maximum 4.3% average latency in the 1458 byte test. A minor upset to this is the IMIX tests, where

[3] `https://www.andrew.cmu.edu/user/aspratt/accelerated_aes/`

the earlier observed pattern is basically reversed. Compared to the baseband measurements in Table 4.2, the cryptography drivers are much faster, which is not surprising due to the much heavier computation requirement of the LDPC processing performed by the baseband drivers.

Table 4.5: Throughput for crypto

Platform	Packet size (B)	Average (Mbps)	Median (Mbps)	Std dev (Mbps)
CPU	243	303.7	303.7	0.09
	486	306.5	306.5	0.09
	1458	308.3	308.3	0.12
	IMIX	308.9	304.8	10.8
GPU	243	365.9	366.8	4.3
	486	633.8	639.2	16.2
	1458	1153	1178.8	47.4
	IMIX	879.6	879.9	1.4

Table 4.5, as expected, reveals that the GPU has a higher throughput for all different packet sizes compared to the CPU. The CPU performs similar results as the GPU for packets sizes of 243 bytes, otherwise, the GPU is significantly better. As before, the results are also better than the results for baseband presented in Table 4.3.

The CUDA implementation was also analyzed with the help of the CUDA tool nvprof, which was described in 4.3.3. The AES-CTR CUDA implementation measured a minimum of 72%, a maximum of 82% and a 75% average of occupancy for IMIX packet distributions which indicates that the AES-CTR implementation utilizes the GPU in a satisfactory way.

4.4.4 DPDK Configurations

With the promise of fast packet handling in the data plane, the overhead of transmitting data around the DPDK application is expected to be negligible. By approaching the scheduling of tasks in two different ways, the aim is to get an understanding of the potential benefit, loss or trade-off in a more flexible two-step implementation versus a more static version with less overhead. This comparison is also mentioned during the introduction in chapter 1. As Figure 4.1, shows, the difference is the middle-step between the two processing tasks.

Table 4.6: Platform latencies

Platform	Packet size (B)	Avg time taken (ms)	Median (ms)	Std dev (ms)
CPU	243	120.38	120.38	0.08
	486	240.54	240.59	0.13
	1458	720.05	720.14	0.71
GPU	243	14.7	14.69	0.03
	486	22.11	22.13	0.14
	1458	27.82	27.4	1.07

Table 4.7: Platform throughput

Platform	Packet size (B)	Avg throughput (Mbps)	Median (Mbps)	Std dev (Mbps)
CPU	243	4.91	4.91	0.003
	486	4.76	4.76	0.002
	1458	4.69	4.69	0.005
GPU	243	43.39	43.42	0.084
	486	55.66	55.62	0.349
	1458	130.86	133.4	6.284

For the dynamic two-step scheduler using C2, specific driver data from Tables 4.2, 4.3, 4.4 and 4.5 are used as a basis for the distribution logic. However, since C1 schedules for running across the entire platform, Tables 4.6 and 4.7 were also produced. While one could have concluded a similar results by adding values from the previously mentioned tables, this extra data also provides a clear basis with included scheduling overhead for the static algorithm running C1.

4.5 Dynamic Batch Aggregation

As previously described by Magheezi et al. [29], dynamic batch aggregation is an important topic to consider in order to deal with low packet rates while maintaining high throughput during heavy load. The persistent kernel seemed a project on its own to implement for the scheduler. The focus instead became finding suitable status data to determine a varying batch size without losing too much throughput. Initially, this was done using packet rates, but as the major point of interest is maintaining latency requirements while maximizing throughput each batch aggregation instead maintained a check on the lowest latency requirement present in the batch, similar to how [36] aggregates their batches.

By predicting an average time between dispatching the batch to the GPU together with the remaining steps of the processing, a threshold is set. The batch is then sent to the GPU kernel, either when the number of packets aggregated has reached the size threshold, or during low loads when a packet is about to expire in terms of latency. Important to note however is that a minimum batch size had to be set. A completely free batch size selection meant sometimes single packets were enqueued, which in turn slowed down the scheduler greatly and cascaded into more minimal batch sizes.

4.6 CUDA Memory Management

The memory transfer of data between CPU and GPU is done in a number of steps. Upon reaching the host function responsible for configuring and calling the kernel, the packet data is stored as pointers in a two-dimensional array, where each pointer points to a packet which in turn contains an array of bytes representing the data.

For the AES-CTR encryption, the open-source kernel used receives the data as a one-dimensional array of bytes. To avoid having to put time and effort into rewriting the kernel and ensure coalesced access, the two-dimensional array is copied to a one-dimensional array. This also ensures contiguous memory storage rather than risking fractured allocations and pointer chasing. This obviously comes at the cost of a copy step. Since the one-dimensional array will only be used on the device, it is explicitly copied to the device memory using the standard `cudaMemCpy`.

The baseband implementation was harder to work with and integrate properly due to the way kernels were set up surrounding 243 MB chunks. However, from a memory point of view, it maintains the same approach as the AES-CTR code.

4.7 Scheduler

The scheduler loop runs on a designated CPU core. On top of running the algorithm distributing the packets among workers, it also handles the RX (receive queue) and TX (transmit queue) of the DPDK application. This means the scheduler is both the entry and exit point of packets processed by the DPDK application. Together with the TX responsibility it also maintains the reorder logic, corresponding to reorder buffers and decisions surrounding those.

There are quite a few internal buffers in the scheduler and the connected workers, meaning there are many occasions where the buffers get full during heavy load. Since packet loss is a very costly event, the scheduler ensures no packets are ever lost, instead errors are treated with retries. This means that, in the event of full buffers, the scheduler stops retrieving incoming packets from external sources until it has been able to handle the packets currently residing in internal buffers. Indirectly, this can lead to packets being lost due to external sources dropping packets which are not accepted in a timely fashion by the scheduler. This would be likely if the scheduler were placed in an environment whose network load exceeds the capacity of the scheduler and its underlying hardware. But since such events are external, the scheduler focus on ensuring no internal packets are lost during its direct responsibility of the data. Below code snippet showcases how this retry logic is maintained for the distribution of packets to the workers. While some things differ, every distribution buffer inside the scheduler utilizes a general version of this approach to ensure zero packets are dropped.

Source Code 4.1: Scheduler distribution code

```
...
if(nb_rx == 0)
{
    nb_rx = rte_ring_dequeue(rx_ring, pkt_array, burst_size);
}
if(nb_rx > 0)
{
    while(idx < nb_rx)
    {
        ret = distribute(worker_id, pkt_array[idx], ...);
        if(ret == 0)
        {
            full = 0;
            idx++;
        }else
            full++;

        if(full == workers)
            break;
    }
    if(idx == nb_rx)
    {
        nb_rx = 0;
        idx = 0;
    }
}
...
```

The variable `nb_rx` stores how many packets were dequeued from the `rx_ring` when calling `rte_ring_dequeue(...)`. These packets are then stored in the `pkt_array`. If any packets are retrieved, `nb_rx` will be greater than 0, and the `while`-loop will iterate over each available worker. If no worker has room for the packet, the condition `full == workers` will terminate the loop, leaving `idx` to continue on the currently held packets without dequeueing a new batch next iteration. Batch sizes, in the code refered to as `burst_size` can vary but is at most 32 packets except for the ring connecting the GPU worker to the GPU driver, which can send larger batches. This value is simply inspired from DPDK example solutions, which always maintains this maximum as an appropriate value for utilizing the ring buffers efficiently.

Source Code 4.2: Ring buffer enqueue code

```
...
static int enqueue(struct packet *m, struct ring_buffer *r,
             struct pre_ring_buffer pkt_array , int burst_size)
{
    int ret = 0, j, len;

    len = pkt_array.len;

    // enough ops to be sent
    if (len >= burst_size)
    {
        ret = rte_ring_enqueue(r, pkt_array, burst_size);

        if (ret == 0)
        {
            return -1;
        }

        len = 0;

        if (ret != burst_size)
        {
            for (j = ret; j < burst_size; j++)
            {
                pkt_array.buffer[len] = pkt_array.buffer[j];
                pkt_array.buffer[j] = NULL;
                len++;
            }
        }
    }

    pkt_array.buffer[len] = m;
    len++;
    pkt_array.len = len;

    return ret;
}
...
```

There is also another buffer ahead before actual enqueuing the packets on the designated ring buffer. This buffer is implemented as a final step before every ring buffer enqueue. There are multiple reasons for this. The first purpose is for handling the event of a full ring buffer by separating the distribution logic from these issues. Another benefit is the flexible choice of burst sizes. These pre-ring buffers are held within a `struct` which also maintains the length of the buffer as a sort of metadata. This length is then used to decide when the array holds a sufficient number of packets before adding them to a ring and to rearrange elements of the buffer if required.

The code snippet in Source Code 4.2 illustrates a slightly simplified version of this solution. There are some specifics depending on what destination the ring is attached to, but that mainly surrounds metadata connected to the packet required for encryption or baseband. Basically, packets are added to the `pkt_array` until they fulfill the amount specified by the `burst_size` argument. This argument, in turn, is decided by the sender, and can, therefore, vary both for different rings but also for the same ring under changing conditions.

The condition checks if the return value of `ret` is not equal to the number of packets enqueued. This means the ring was too full to enqueue all packets. For such an event, the array is rearranged, moving the pointers of packets which were not successfully enqueued to the front of the array and adjusting the length accordingly.

4.7.1 Reorder Implementation

As shown in Table 4.1, packets of type 1 and 4 have to be reordered before being transmitted through the TX. The scheduler uses the sliding window technique to decide if a packet is late, early or on time.

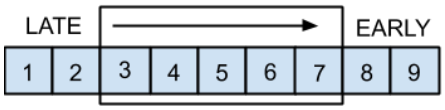


Figure 4.4: Sliding window example

The window has a pre-set size in which it reorders packets if needed to make sure they are transmitted in the right order. As shown in Figure 4.4, packets that arrive before the window are considered late while the packets that arrive after the window are considered early. As soon as packets have arrived and been ordered in the correct order, as many packets as possible are transmitted and the window slides further up the packet numbers.

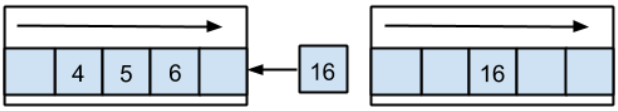


Figure 4.5: Packet arrives early example

Figure 4.5 illustrates what happens when a packet arrives too early. In this example, the scheduler is still waiting for packet 3 to arrive before it can transmit packet 4,5 and 6. But before packet 3 arrives, packet 16 comes. When this happens, the scheduler drains and transmits the packets inside the window before the window slides further ahead and sets packet 16 as the midpoint of the window. Early packets can happen since due to the underlying heterogeneous multi-core system on which the scheduler operates where different cores can process packets independently at varying speeds.

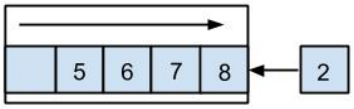


Figure 4.6: Packet arrives late example

Figure 4.6 illustrates what happens when a packet arrives too late. The scheduler is waiting for packet 4 which should be in the front of the window. Instead, packet 2 arrives but since the window edge is located at 4, packet 2 is considered late. When this happens, packet 2 is transmitted directly to make sure that it does not arrive even later than what it already is.

4.8 Round-Robin Algorithms

As described in Section 2.4, the round-robin algorithm tries to balance the load between the available workers. Neither packet size, reordering or any other available information is taken into consideration for the round-robin algorithm, only if a worker is available for more work matters. Since the GPU has better throughput and latency for both the baseband and crypto drivers, it is expected that the round-robin algorithm will distribute most packets to the GPU. Based on the measurements in Table 4.2, the GPU worker will be assigned more than 11 times more packets than the CPU workers combined, at least during heavy load of packets. The only difference between the two round-robin configurations is that the C1 round-robin algorithm only makes one decision for each packet, if it should be sent to the GPU or one of the CPU workers. The C2 round-robin must do two decisions for each packet.

4.9 Packet-size and latency sensitive scheduling

The distribution functions, responsible for the actual scheduling of packets between workers, are based on previously measured data presented in Tables 4.2-4.7. These functions are called for every packet that enters the scheduler. Below follows an in-depth description of these algorithms and the differences between configurations.

4.9.1 Configuration 1

The distribution decision for C1 is made once for each packet at the beginning, following the packets entrance to the scheduler.

Algorithm 1 C1 Scheduler distribution

input : packet size, packet size threshold for cpu, packet latency, packet latency threshold for cpu, input rate, input rate threshold for cpu, if reorder is needed and worker ids
output : 0 if successfully enqueued on worker, otherwise 1

```
1: if pkt_size ⩽ cpu_pkt_threshold && latency > cpu_latency_threshold && input_rate >
   rate_threshold && !reorder then
2:     ret ← enqueue_packet(cpu_worker);          // returns 0 on success or 1 if the CPU input buffer is full
3: else
4:     ret ← enqueue_packet(gpu_worker);          // returns 0 on success or 1 if the GPU input buffer is full
5: end if
6: if ret == 0 then
7:     return 0;                                  // successfully enqueued packet on priority worker
8: end if
9: if ret == 1 && (latency < gpu_latency_threshold || reorder) then
10:    return 1;                                  // low latency requirements should be retried to the faster processing of the GPU
11: end if
12: return enqueue_packet(worker_id);
```

As Algorithm 1 shows, packet size, reordering, input rate, and latency requirements serve as the decision points for the distribution. This is because they are all common properties of data traffic, and in a heterogeneous system consisting of a CPU and GPU, they have a distinct impact on performance.

In order to determine suitable threshold values, measurements were done to evaluate how long time the GPU and the CPU required to process a given batch of packets. The latencies for each packets are measured by tracking the packet during its lifetime inside the scheduler. Upon entry it recieves a timestamp, which is inserted into available fields of the `rte_mbuf` struct. Whenever the scheduler wants to determine the packets current latency it extracts the packets timestamp and compares it to the current time. In Tables 4.6 and 4.7 average latency and throughput values for processing different packet sizes on the two platforms are shown.

The heavy computation of LDPC processing is the main reason why the GPU outperforms the CPU for all packet sizes. The ability to parallelize the computation of packets, even across smaller 243 MB data sizes, provides a speedup of 8 times the CPU latency despite the overhead of kernel launches, data transfer, and batch aggregation. The baseband processing is the major bottleneck in performance, especially when utilizing the CPU and therefore the threshold and decisions are practically made based on baseband performance.

While any packet is faster on the GPU when LDPC is performed, the overall throughput of the scheduler would be reduced by simply never utilizing the CPU workers during heavy traffic. Therefore the scheduler first checks the packet properties for packets which it considers applicable to the CPU, meaning they can be offloaded to a CPU worker without risking latency misses or extremely long processing times.

As a result, any packet of the smaller size and no reorder together with non-existent or very high latency requirement is prioritized to the CPU for offloading. All other packets are first attempted to be enqueued to the GPU. This prioritization is aimed to make the GPU available to a greater extent for high priority packets with low latency requirements.

However, if the GPU is full, a final check is made to ensure that the denied packet does not have any critical requirement which the CPU is not believed to be able to fulfill. Noteworthy is that reordering is considered such a property because, while reordering itself is not necessarily critically latency bound, latency sensitive packets might demand reordering. Generally, it is also preferred to keep reorder required flow on the same worker since spreading it out risks causing unnecessary wait

times and order misses, especially in a heterogeneous environment. To ensure no packets bound for reorder will serve as a bottleneck for other latency sensitive packets, all reorder packets are sent to the GPU for more predictable behavior. If this is the case, the scheduler retries for the GPU. If not, the scheduler enqueues the packet to a fallback CPU worker.

4.9.2 Configuration 2

The scheduler running with C2 makes a scheduling decision once for every processing task, meaning the packets can change platform between tasks.

Algorithm 2 C2 Scheduler baseband distribution

input : threshold values, worker ids and packet meta data
output : 0 if successfully enqueued on worker, otherwise 1

```
1: return enqueue_packet(gpu_worker);
```

Algorithm 3 C2 Scheduler crypto distribution

input : input rate, minimum, medium and maximum input rate threshold, if reorder is needed and worker ids
output : 0 if successfully enqueued on worker, otherwise 1

```
1: if input_rate > rate_threshold_max && reorder && then
2:     ret ← enqueue_packet(reorder_cpu_worker_id); // returns 0 on success or 1 if the CPU input buffer is full
3: else if input_rate > rate_threshold_medium && !reorder then
4:     ret ← enqueue_packet(cpu_worker);          // returns 0 on success or 1 if the CPU input buffer is full
5: else if input_rate > rate_threshold_min && !reorder && pkt_size < cpu_pkt_threshold then
6:     ret ← enqueue_packet(cpu_worker);          // returns 0 on success or 1 if the CPU input buffer is full
7: else
8:     ret ← enqueue_packet(gpu_worker);          // returns 0 on success or 1 if the GPU input buffer is full
9: end if
10: if ret == 0 then
11:     return 0;                                  // successfully enqueued packet on priority worker
12: end if
13: if ret == 1 then
14:     if input_rate < rate_threshold_min then
15:         return 1;                              // if reorder is required and input rate not enough to fully overload GPU, retry on GPU
16:     else if input_rate > rate_threshold_medium && input_rate < rate_threshold_max
        && reorder) then
17:         return 1;                              // if input rate is low enough to allow both crypto and baseband on GPU without overload, retry GPU
18:     end if
19: end if
20: return enqueue_packet(worker_id);
```

Both the baseband and crypto distribution logic is shown in Algorithm 2 and 3. Firstly note the very simple decision for baseband. This is because for the C2 context it is possible to perform encryption on the CPU without baseband. As shown in 4.2 and 4.4 and discussed in earlier sections, baseband benefits greatly from being dispatched to the GPU compared to CPU while the lighter computing of AES-CTR makes CPU encryption more competitive.

This distinct result means there is a potential benefit in ensuring all LDPC processing is made on the GPU, while CPU offloading is tasked with encryption. Therefore all baseband processing is dispatched to the GPU.

For the second processing step, it is important to note that the baseband is the bottleneck to such an extent that CPU encryption is also faster than GPU baseband. However, the average latency is still improved for GPU encryption for almost all packet sizes compared to the CPU. This results in a decision process which attempts to map ranges of input rates to a level of GPU saturation which in turn allows a certain amount of encryption to be dispatched to the GPU.

Table 4.8: Condition explanation

Line number	**Decision**
3	The maximum rate threshold is an approximation of when the GPU baseband is under maximum load. At this point even reorder bound packets are enqueued on the CPU to fully utilize the GPU for the heavy baseband processing. However, they are still prioritized to a designated CPU worker to ensure order success
5	For a rather high load, but a step below the maximum, packets without reordering are all considered applicable to the CPU. This means only order required packets are prioritized to the GPU.
7	The final condition checks, on top of reorder, also packet size. If the size is considered small it allows CPU distribution. This rate threshold attempts to capture the level where the load is low enough to almost allow both crypto and baseband to be processed on the GPU without throttling. Below this threshold no CPU processing is made since the input rate is considered low enough to only use the GPU.
16	As a follow-up from line 7, this condition simply ensures that during such low input rates the scheduler does never fallback to the CPU. It might be excessive, since the GPU should never be full during this occasion anyway, but during varying traffic conditions it can help.
18	A guard to ensure no reorder packets are enqueued on the CPU during a medium traffic load.

These thresholds are then treated with packet property prioritization, which is further explained in Table 4.8. Line numbers correspond to the line of the condition in Algorithm 3.

4.10 Machine Learning Algorithm

With a reinforcement learning approach, the ambition is to provide the algorithm with sufficient information about the schedulers state to aggregate data for smart decisions. Since the desire is for the algorithm to learn and provide insight, the implementation does not try to steer the result using custom crafted special case rewards distinguishable related to distributions and specific test suites. This means state information and reward functions are attempted to be kept at a general level, which is also important for the feasibility of the training of the algorithm and how adaptive it is to change in environment and resources.

4.10.1 State space

The state space S of a reinforcement algorithm grows rapidly with any additional state information due to the combinatorial explosion of each possible state, which in turn is a consequence of the curse of dimensionality [44]. Therefore great caution and consideration has to be put into concluding which information is to be represented as states. Some relevant states for the scheduler surrounds the state of the buffers, packet information, and traffic load. The most straight-forward information is packet types. As mentioned in Section 4.2.2, the scheduler maintains a table of types mapping to different packet properties. These types are one of the basic decision points and are therefore provided to the algorithm where each type represent a state. Buffer status and traffic load are also considered, but since keeping one state for each possible value of input rate or the number of packets in buffers would be extremely inefficient, this information is transformed into discrete representations. The state index is therefore calculated according to

$$S = I * (5 * (workers^2)) + RS * 5 + PT \tag{4.1}$$

where I is the current discretely represented input rate, RS the discretely represented buffer status of the workers and PT the mapped type of the packet. The discrete range of $I = [0,4]$ and RS is calculated as seen in below code snippet. This means it varies based on the number of workers connected.

Source Code 4.3: Discrete buffer status

```
int discrete_representation = 0;
int total_load = 0;

for(int j = 0; j < worker; j++)
{
    for(int k = 0; k < enqueue_rings[j]; k++)
    {
        total_load = ring_elements_count[k];
    }

    if(total_load > load_threshold)
    {
        discrete_representation += 2^j
    }
}
```

Both C1 and C2 use the same state-space, but to clarify, C2 gathers the buffer status from both enqueue stages since those together represent the load on the worker.

4.10.2 Actions

Each state space is attached to an action A, which will transfer the scheduler to new states and yield rewards as feedback for the action given the state space. For the scheduler, these actions simply correspond to the decision of where the packets are enqueued.

Since the main difference between C1 and C2 is that one schedules based only on the platform, while the other schedules packets to individual tasks, there is some difference in action setup. C1 either schedules tasks on the CPU or GPU, which represented numerically results in a value range of $A = [0,1]$.

C2, however, considers AES-CTR or LDPC processing. Therefore the setup for C2 results in four action choices. This means the number of actions for C2 is represented numerically as $A = [0,3]$. However, since the order of processing is determined–baseband first then encryption–the action decisions are separated in two steps. That results in a smaller action space of 4 actions, rather than 8. Both steps use the same state space.

4.10.3 Reward

The reward for each action with a corresponding state comes with the challenge of ensuring state information and actions produce useful feedback. It is a matter of concluding what information can be extracted following a task, and also correctly tie it to the corresponding action. The most intuitive aspect which the reward function consider is the average latency of the packets scheduled for a task given the active state. This is easy to measure and tie to the correct state and action. It is also probably the most important value to consider when rewarding task choices. However, on top of the latency, other factors are harder to take into consideration. Throughput would be a desirable result to consider, but since it is the result of the overall state of the application, it is harder to tie it to the beneficial action.

The throughput also depends on the input rate, which would require training the algorithm to be done in steps of static traffic load. Nevertheless, it is an important weight to measure in order to provide some sort of trade-off to simply minimizing latency, since this could otherwise possibly be cheated by simply maintaining a very low input rate. Therefore the throughput is measured as a reward, with the belief that despite the risk of misfired rewards, it can still distinguish favorable actions in the long run.

Another important feature of the scheduler is the fulfillment of latency sensitive requirements. This is a relevant decision point for any scheduler which process packets of different priority and hard demands, for example, voice or live video transmissions. The percentile success rate of latency sensitive packets is therefore also added to the reward equation.

When finally designing the reward function for the aggregated results a neutral and intuitive stance was desired. While weighing each result individually could possibly provide a better result, such an approach would require a lot more time in iterations, as well as risking a too customized and environment-bound solution. With the delimitation of time the goal is also not necessarily to provide the optimized solution, but rather evaluate the feasibility of a intuitive and adaptive one.

Therefore, all properties are factored together to provide a value where significant variations in any property always yields major consequences. This in turn does put a higher requirement of well chosen states included in the reward function, since values which would be considered less important would perhaps have an undesirably large influence on the result. However, with this in mind the previous states described are still included, since they are considered important enough to always affect the schedulers behaviour.

For both C1 and C2, the reward function is as follows

$$R = \frac{LR_s}{LR_t} * \frac{T}{L} \tag{4.2}$$

where LR_s is the number of achieved latency requirements, LR_t total number of latency sensitive packets, T average throughput and L average latency. This reward is then added to the correspond-

ing action and state from which the data was extracted according to equation 4.3 where a is the number of actions available to the algorithm

$$Q\_table[S * a + A] = R; \tag{4.3}$$

4.10.4 Training

Compared to the static scheduler, the ML-scheduler has some additional function calls to an added file maintaining all algorithm functionality. These functions correspond to choosing actions, maintaining information about which packet connects to which action and state and finally provide rewards. Basically, it acts as a library providing an API.

Training the algorithm is done through choosing actions in a random manner for distribution, and record the choice through a table of active packets about to be processed. This table holds information about in which state the chosen action was taken. Once a packet is processed, the results are connected to the correct packet and stored in a separate state table, where results for each action-state is stored. These results are then aggregated over the course of a training session, and once complete each state-action is given a reward based on the results. Finally, these rewards are stored in a separate file for future use and modification.

The ML code also runs on the same core as the scheduler, which means some extra overhead compared to running on a separate core. Although this is only significant during training. During actual runtime, it was considered minor enough to be negligible. An overview of the design can be seen in Figure 4.7 and 4.8, showing at which points the machine-learning implementation connects to the scheduler for C1 and C2 respectively.

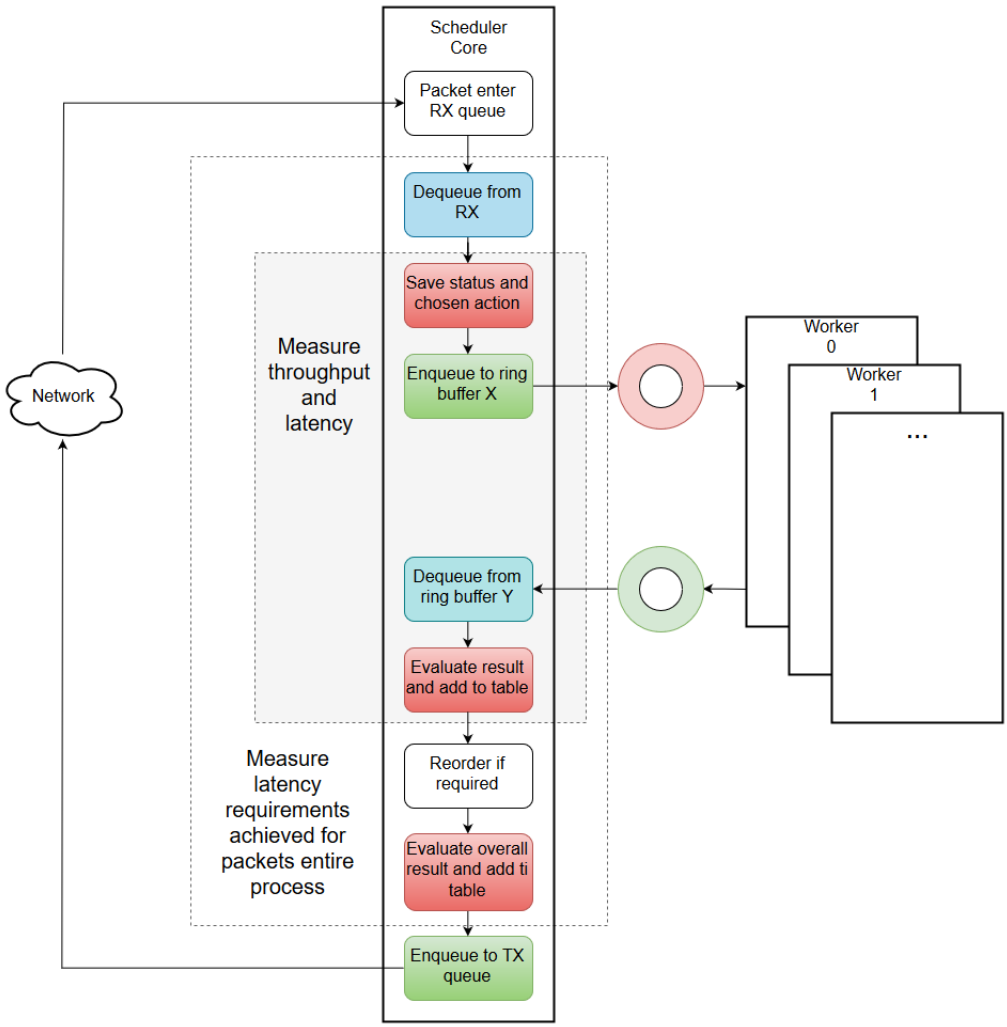


Figure 4.7: ML C1 design overview

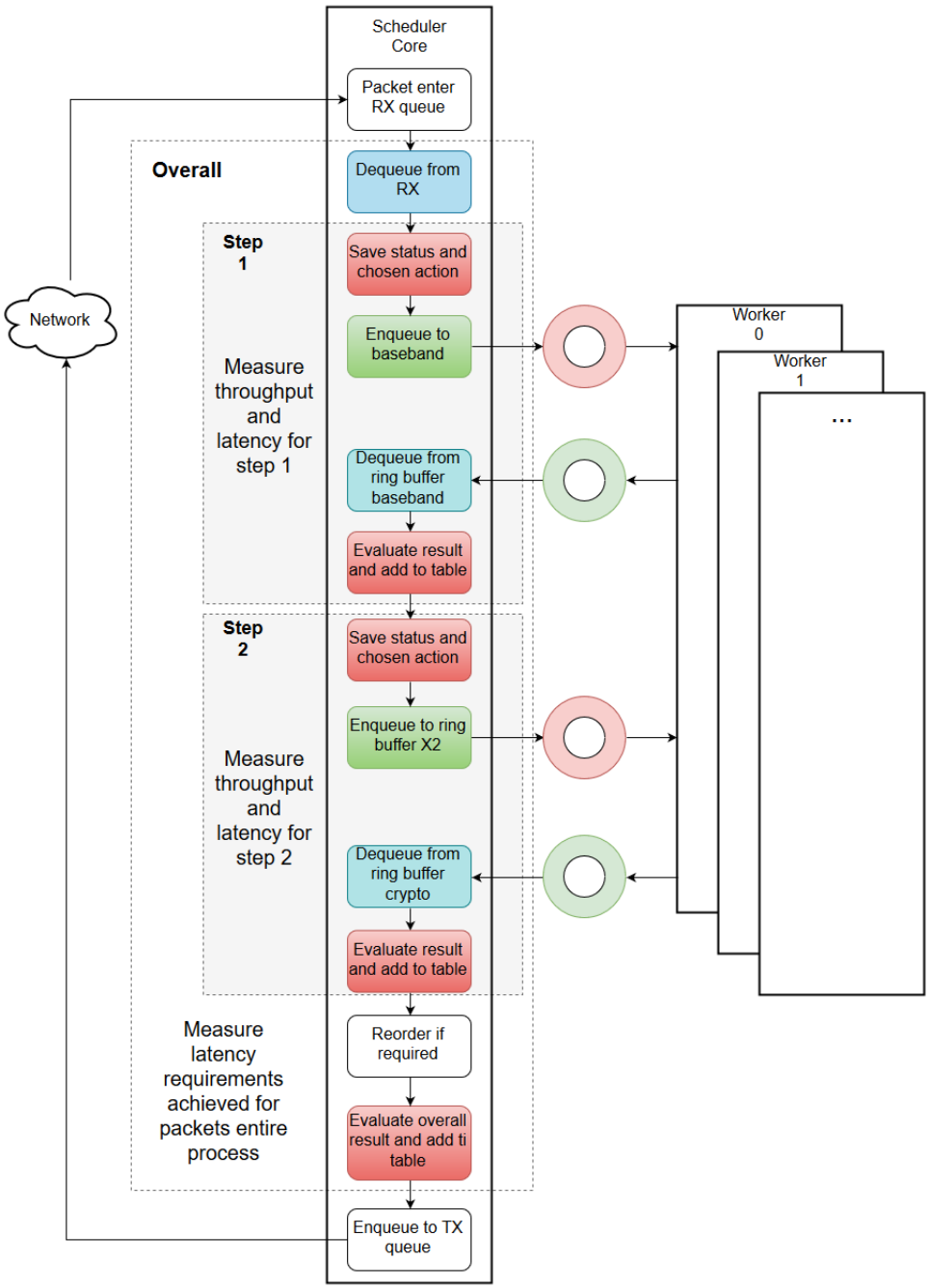


Figure 4.8: ML C2 design overview

The C1 implementation evaluates the entire processing chain. For C2, two independent decisions are evaluated for each task. However, as mentioned in Section 4.10.1, they indirectly consider each other through an overall state space.

For both C1 and C2 the evaluation is split in a processing step and an overall step stretching across the entire scheduler including overhead such as reordering. Latency and throughput are measured local to the processing, but since latency achievement is not concluded until the packet actually leaves the scheduler this requires another final step. This final step also means the impact of reordering is indirectly considered.

The API of functions are presented as pseudo code in Source Code 4.4 with their relative location in the scheduler.

Source Code 4.4: ML API

```
...

//dequeue from RX

if(training && random_action)
{
    random_action();
}else
{
    choose_action(packet_type, ring_status, rate, step)
}

if(training)
{
    //status of scheduler during time of action is saved
    add_action_to_table(action, packet_type,
        unique_id, ring_status,
        rate, step, time_begin,
        current_throughput);
}

//enqueue to worker
...
//dequeue from worker

if(training)
{
    ml_learn_processing(unique_id, step,
        time_end, current_throughput);
}

//for C2 a second iteration follows here

//final tasks such as reordering

if(training)
{
    ml_learn_overall(unique_id, latency_requirement)
}

//enqueue to TX
```

4.10.5 Learning

Once a training file has been generated, the scheduler can load the file contents into a table. This table simply holds a reward value for each state and action. Whenever a packet is to be distributed, the scheduler sends the information surrounding the applications state - which is collected the

same way as during training - to a machine learning function. This function then simply calculates the state value given the state information of the scheduler and chooses the action with the highest reward value for the state.

4.11 Test Application

To be able to evaluate our application, a test application was developed. For simplicity, it was also made using DPDK, but it runs as a separate process. This means the application has access to the same DPDK allocated system memory as the scheduler, allowing utilization of DPDK shared resources such as memory pools and ring buffers. Since the scheduler is the persistent program in this scenario, it creates all buffers and memory pools upon startup. The test application then retrieves these structures through memory lookups using unique `char[]` names as identifiers. These object names are shared between the scheduler and test application through a shared `.h` file. A basic step-by-step overview of the test applications workflow is shown below.

1. Allocate bulk of `rte_mbuf` objects representing packets from a DPDK memory pool
2. Modify packets in terms of type and size
3. Enqueue the bulk of packets on the scheduler input ring buffer
4. Poll scheduler output ring buffer for any processed packets.
5. Free processed packets by returning them to the DPDK memory pool for future reuse.

The test application is started with a set of command-line arguments, or set to default values if none is specified. These arguments correspond to a couple of options where the important ones are testing size, in terms of packets to process, or duration and packet sizes with corresponding distributions. The IMIX distribution is configured with a set of sizes and a corresponding share of the total test size, which is relative to the other packet size shares. For example, the size set 243,486,1458 with shares corresponding to 2,1,2 should result in an approximate packet size distribution of 40% 243 bytes, 20% 486 bytes and 40% 1458 bytes.

The modification of the packet type mentioned in step 2 of the bullet point list above corresponds to packet properties such as latency and reordering requirements. Depending on which packet type is set for the `rte_mbuf` the scheduler will map it to different requirements upon entry to the scheduler program, and schedule it accordingly.

A test run is completed when the set test size of packets or duration is reached. A packet is considered processed when it reaches the test application through the scheduler output ring buffer. Once the test condition is fulfilled, the test application stops allocating and sending packets. After quitting the working loop it uses the number of packets and bytes processed to calculate the throughput in terms of gigabytes per second. It also aggregates the time taken, or latency value, from each processed packet and calculates the average packet latency across the test run.

To be able to fully evaluate the schedulers, the test application has the possibility to perform four different tests which correlate to four different scenarios providing interesting feedback of the scheduler's performance under reasonable conditions. The four tests are a *low load test*, a *medium load test*, a *heavy load stress-test*, and a *standard load test*. The low load is especially interesting for a system utilizing a GPU with the desired aggregation of larger data chunks. The medium is more or less a middle-step and could simply consider performance during an average load where resources are utilized, but not maxed. The stress test concludes and evaluates maximum throughput and performance during such conditions.

The standard test is used to simulate a scenario of varying conditions and the schedulers adaptation to these, as well as performance when traversing different thresholds. This is probably the most realistic simulation of a real world network node and should conclude how well the algorithms can handle a diverse traffic environment. The source code for the standard test trace file is created is shown in snippet 4.5

Source Code 4.5: Trace for Standard test

```
uint32_t std_trace[30] = {5,5,5,6,6,7,8,10,12,15,
                          12,8,4,4,4,4,5,7,6,4,
                          5,5,6,7,8,8,8,10,12,14};
int j, k;
for(j=0; j<500; j++)
{
    k = ((int)floor(j/10))%30;
    trace[j] = 100*std_trace[k];
}
```

Finally, the test application can also generate a random trace using the clock as a seed to generate pseudo-random numbers. It provides a random rate as well as a random order of packet sizes, but the overall distribution remains approximately the same as the other test suites. This feature is mainly used for training the machine learning algorithms.

4.12 Evaluation

The evaluation of the scheduler is mainly done through the test application, which tracks a number of factors. However, some data is collected inside the scheduler due to the inconvenience of transmitting it to the test application. The metrics aggregated over a test run are:

- Average packet latency (s)
- Throughput (Gbps)
- Latency sensitive packet requirements achieved (% of number of packets having such requirements)
- Packet distribution between platforms (%)
- Succesful reordering

Per packet data such as average latency, latency requirement success and reordering are gathered inside the scheduler and transmitted to the test application through the usage of the `rte_mbuf` meta fields[4]. Timestamps are taken at the entry of the scheduler for each packet, and stored in, where `m` is a `struct *rte_mbuf`, the `m->timestamp` field. The other two results did not have a proper field specifically for this purpose, but there are plenty of other members who are not utilized by the scheduler. For both successful latency requirement and reordering an arbitrary available and non-busy member field of the `rte_mbuf` field, which had a satisfying type, were used, simply setting a 1 for success and 0 for failure. These are then extracted from each packet that returns to the test application. On top of an overall result for the test run, a trace-file is also generated. This

[4] https://doc.dpdk.org/api/structrte__mbuf.html

file simply splits the test run into second long intervals and gathers the rate of packets sent to the scheduler, average latency, latency achieved and reorder during this interval.

Throughput is simply calculated by the test application by adding up all packets with their respective size when they arrive from the scheduler. Distribution is however only measured inside the scheduler, since tracking the stats for each worker results in a few too many data fields to bother transmitting them back to the test application.

For each test, an IMIX distribution will be used, which is the same IMIX as referred to in earlier sections presenting driver performance. As earlier motivations stated, this distribution of packet sizes is based on the distributions retrieved from the internet exchange points shown in Figure 2.5 and 2.6.

4.12.1 Power Consumption

To measure the power usage of the platform software provided by the Jetson Xavier was utilized. The module provides 3-channel INA3221 monitors which can be accessed through two I2C addresses where each channel refers to a rail which corresponds to a part of the Jetson Xavier system. Details on how these addresses are set up and more can be found in the design guide provided by Nvidia [45]. Basically a current power consumption value, in terms of mW, is retrieved by calling the designated system address as this example shows:

- `Power (mW): ../<address>/iio_device/in_power<Channel>_input`

By switching address and channels, all parts of the system can be measured. This is in turn done through a simple bash script which copies the values held at each of the addresses corresponding to the GPU, CPU, SOC and DDRAM, adding up to the total consumption. The script runs alongside the test application accessing these addresses once every second. The interval is kept at a second to avoid excessive usage of CPU resources for these reading tasks.

5 Results

In this chapter, the results from the test application for the schedulers will be presented. The results will not be discussed or analyzed in this chapter, this is done in Chapter 6. All figures that include the input rate have the input rate measurement, in terms of packets per second, to the left and the property that the schedulers are measured by to the right. All tests are run on a Jetson Xavier, using one GPU worker and two CPU workers. All four test suites are presented one at a time, where the two configurations C1 and C2 are shown in a comparable fashion next to each other.

5.1 Overhead Measurements

For each test, the overhead will be measured at three different spots for each scheduler, as shown in Figure 5.1. Overhead 1 is the same for both configurations and includes the scheduling and enqueueing of packets to the baseband device. Overhead 2 have some differentiation between the two configurations. For the C1 schedulers, it includes the dequeuing of packets from the baseband device and the enqueuing of packets to the crypto device. On top of the steps made by C1, C2 also includes the process of enqueueing and dequeuing back the packets to the scheduler core. The scheduler then makes a decision for each packet and schedule it to one of the workers. The worker core then dequeues the packet before doing the same as the C1 schedulers, enqueuing it to the crypto device. Overhead 3 is the same for all the schedulers and measures the time from when a packet has been dequeued from the crypto device, gone through the reordering if required, and then finally enqueued to the TX queue.

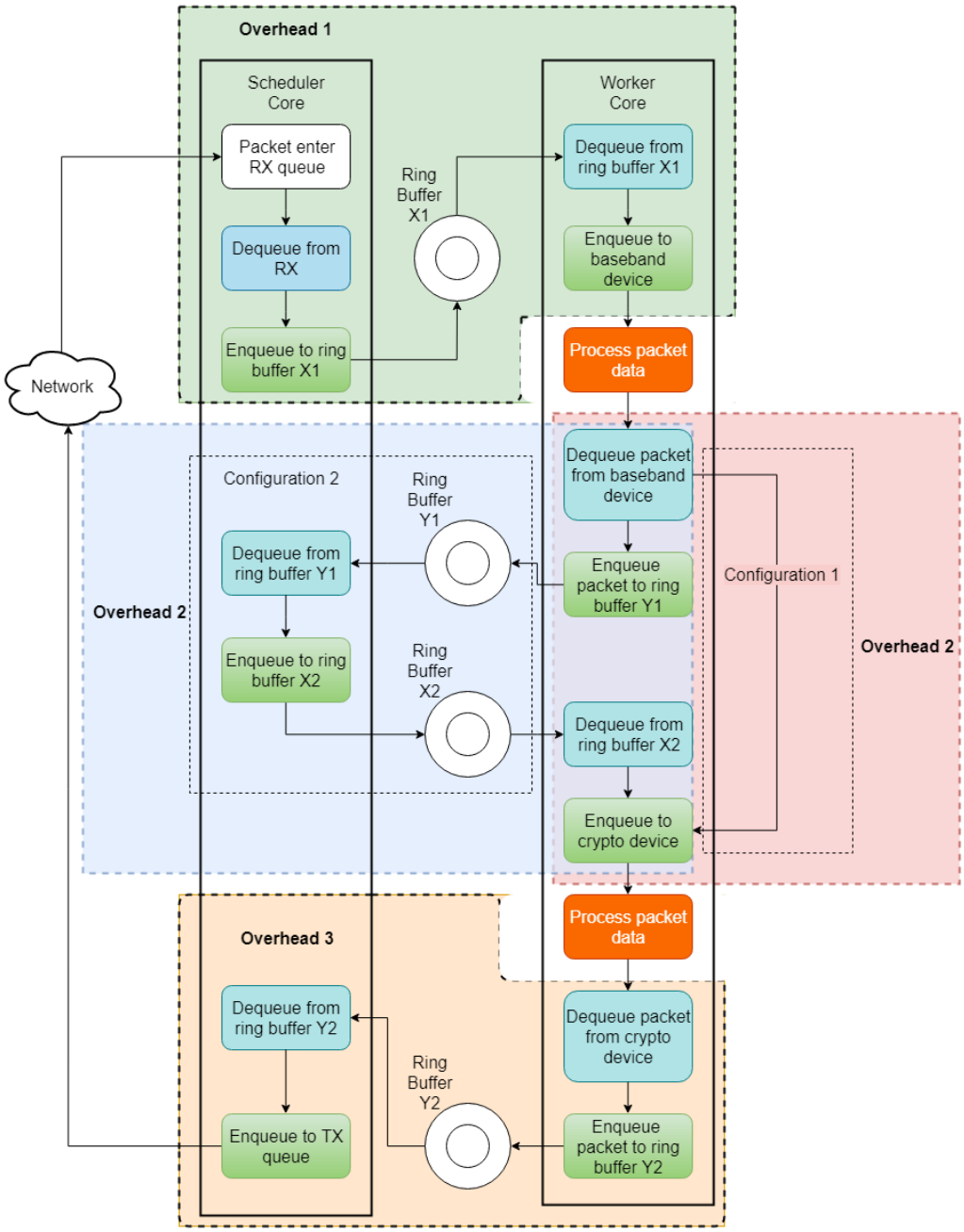
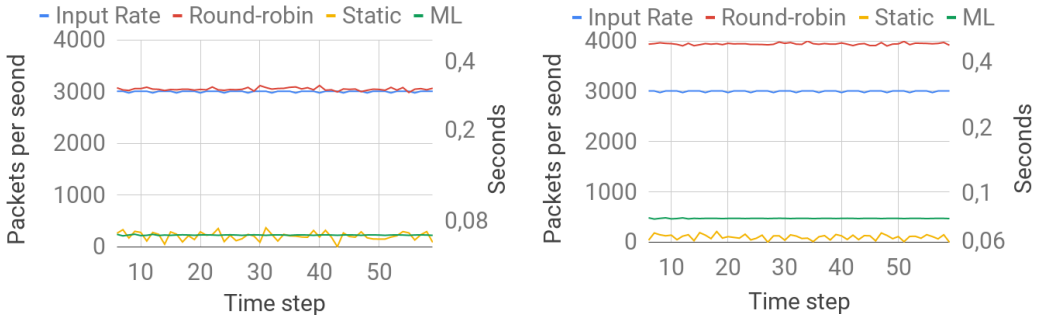


Figure 5.1: Simple packet flow with overhead

5.2 Low Test



(a) Configuration 1

(b) Configuration 2

Figure 5.2: Low input test with continuous average latencies

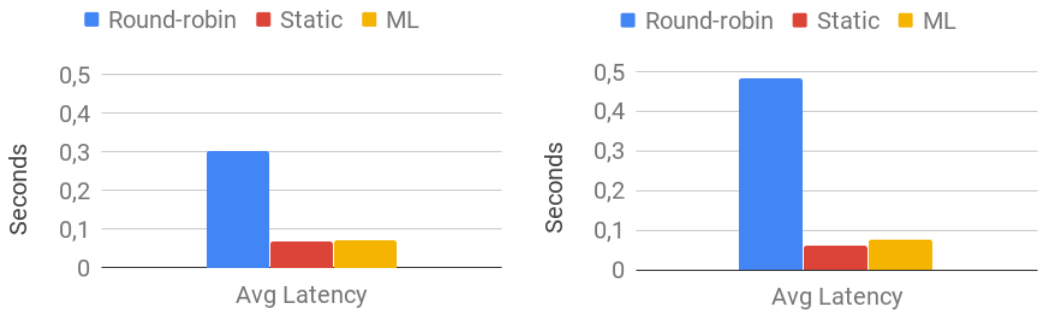
Figure 5.2 shows that the latency is significantly higher for the round-robin schedulers than for the machine learning and static schedulers. Between the static and machine learning schedulers, the C2 machine learning scheduler had the highest latency and the C2 static scheduler had the lowest latency. As can be seen in the figure, the machine learning schedulers have an even latency while the static schedulers have a more varying latency.

(a) Configuration 1

(b) Configuration 2

Figure 5.3: Missed latency-sensitive packets during low input test

As can be seen in Figure 5.3, the schedulers have a hard time to deliver the latency-sensitive packets on time. The C1 static and C1 machine learning schedulers missed about 25%, the C2 static missed slightly more than 25% of the packets, and the C2 machine learning scheduler missed a little more than 75%. Both the round-robin schedulers missed 100% of the latency sensitive packets. When the input rate is as low as 3000 packets per second, it takes a significant amount of time to fill up the GPU buffers which lead to longer waiting times for the packets.



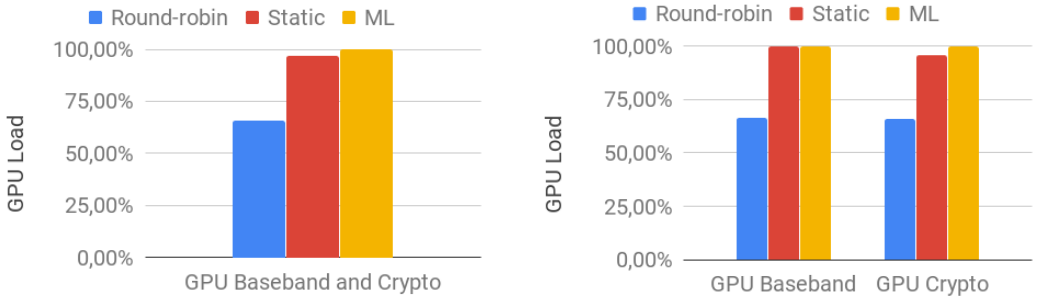
(a) Configuration 1

(b) Configuration 2

Figure 5.4: Average latency during low input test

Figure 5.4 shows that the average latency is highest for the C2 round-robin scheduler and that the C2 static scheduler performed best with 0.062 seconds average latency. The C1 static scheduler has a 0.068 seconds average latency, the C1 machine learning scheduler have 0.069 seconds average latency and the C2 machine learning scheduler have a 0.076 seconds average latency.

It is important to notice that the difference between the best and worst scheduler when it comes to average throughput for the low input test only differs with 0,084 Mbps. This is simply because all schedulers can deal with the lower input rates. Therefore no real conclusions are drawn based on throughput for these tests.

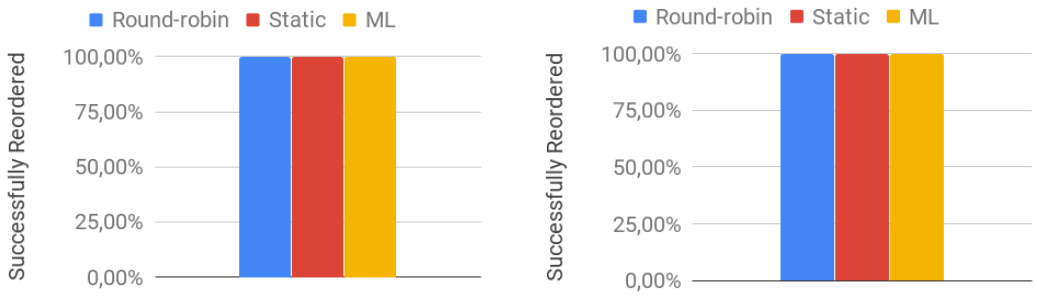


(a) Configuration 1

(b) Configuration 2

Figure 5.5: Average GPU load during low input test

The machine learning schedulers and static schedulers schedule all or almost all the packets to the GPU, as shown in Figure 5.5. The round-robin scheduler schedule approximately 66% to the GPU and this might be the reason why the round-robin performs so much worse with regards to the latency shown in Figure 5.2 and 5.4.

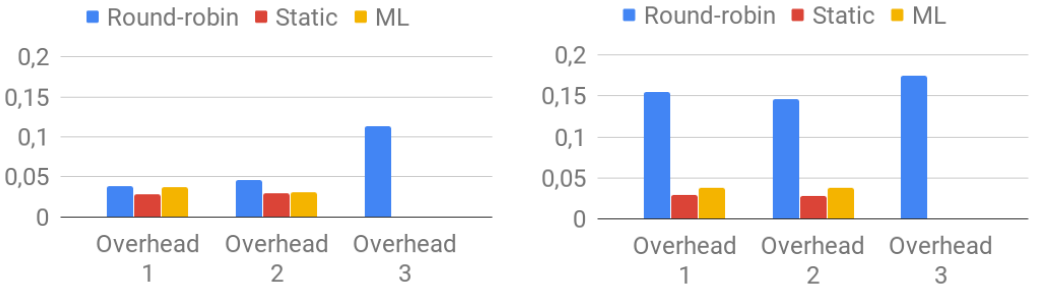


(a) Configuration 1

(b) Configuration 2

Figure 5.6: Packets successfully reordered during low input test

Figure 5.6 show that all the schedulers could handle the low input rate good when it comes to re-ordering packets.



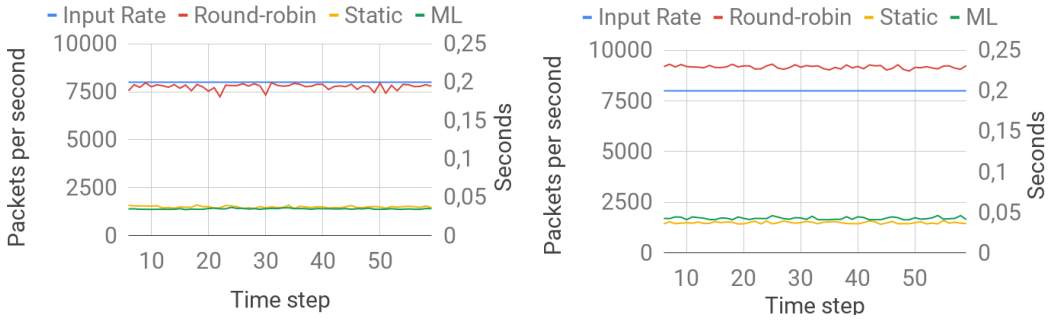
(a) Configuration 1

(b) Configuration 2

Figure 5.7: Overhead during low input test

The C2 round-robin scheduler has much more overhead than the other schedulers, even compared to the C1 round-robin scheduler. Figure 5.7 also show that the overhead for the scheduling between baseband and crypto in C2 is insignificantly higher than that for the C1 schedulers.

5.3 Medium Test

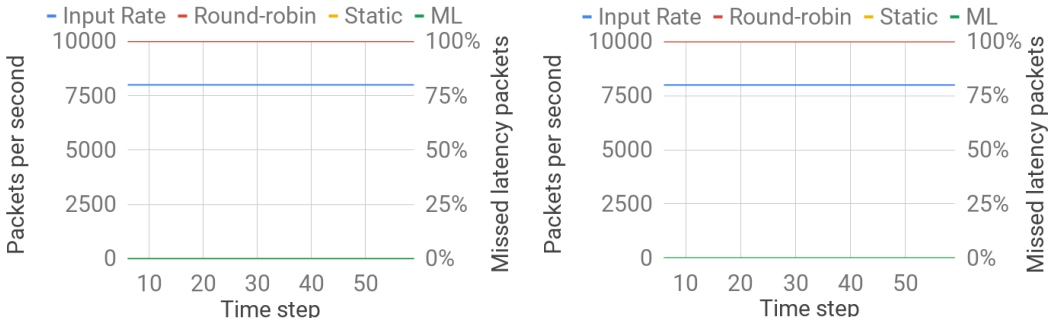


(a) Configuration 1

(b) Configuration 2

Figure 5.8: Medium input test with continuous average latencies

The machine learning and static schedulers performed very similar latencies for the medium input test. The high latency of the C2 round-robin scheduler is noteworthy because it is so much higher than the C1 round-robin scheduler.

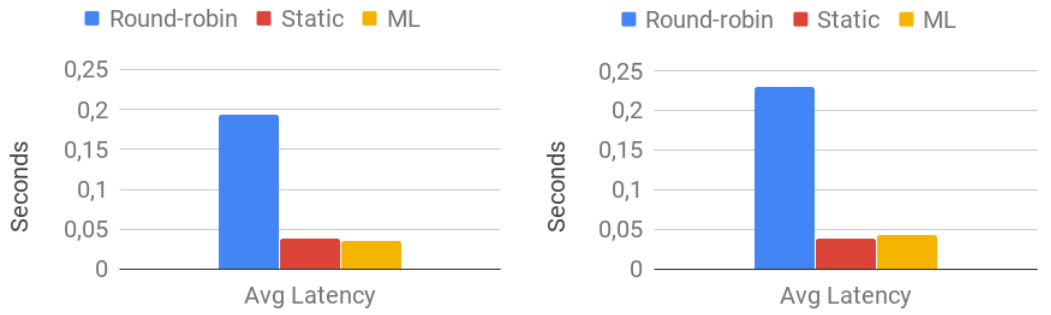


(a) Configuration 1

(b) Configuration 2

Figure 5.9: Missed latency-sensitive packets during medium input test

Neither the static or machine learning schedulers missed any latency-sensitive packet during the medium input test, as shown in Figure 5.9. Both the round-robin schedulers missed 100% of the latency-sensitive packets.

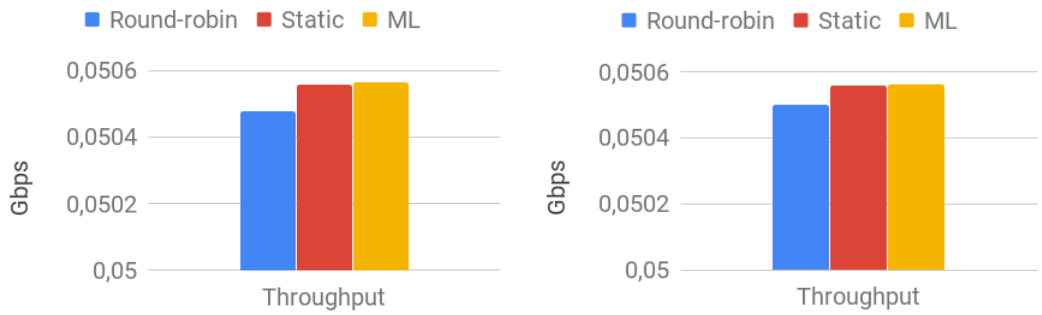


(a) Configuration 1

(b) Configuration 2

Figure 5.10: Average latency during medium input test

Figure 5.10 show that the C1 machine learning scheduler performed the best with regards to the average latency with a measured 0.035 seconds latency, while the C2 round-robin performed the worst. Except for the round-robin schedulers, the C2 machine learning scheduler had the highest latency with 0.043 seconds latency.

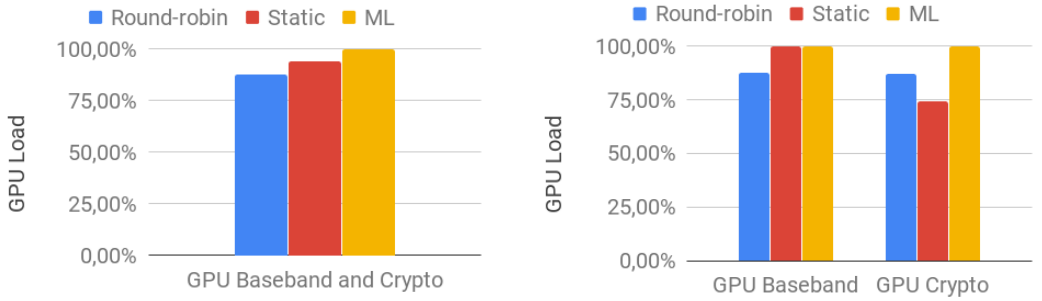


(a) Configuration 1

(b) Configuration 2

Figure 5.11: Average throughput during medium input test

The machine learning and static schedulers performed almost the exact same throughput with approximately 0.05056 Gbps while the round-robin schedulers performed slightly worse with 0.0505 Gbps throughput, as shown in Figure 5.11. This is a difference of 0.06 Mbps or 0.12% in higher average throughput to the advantage for the static and machine learning schedulers.

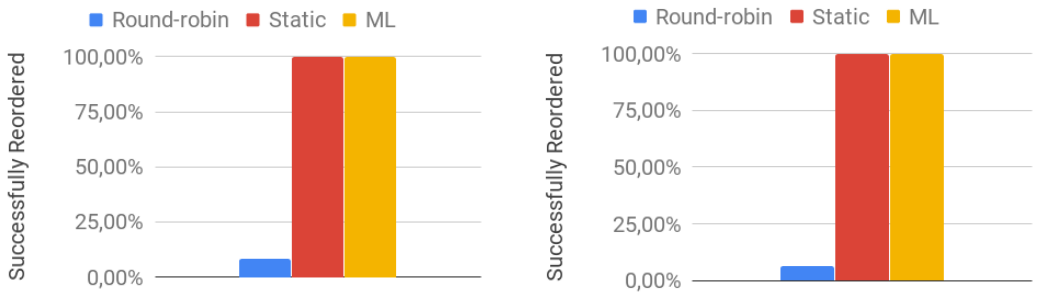


(a) Configuration 1

(b) Configuration 2

Figure 5.12: Average GPU load during medium input test

As shown in Figure 5.12, the C1, and C2 machine learning schedulers schedule all the packets through the GPU. The C1 static scheduler schedule 94% of the packets the GPU and offload just 6% to the CPU. The C2 static scheduler performs all the LDPC decoding on the GPU but for an input rate if 8000 packets per second, it offloads 26% of the encryption to the CPU and performs the rest on the GPU. The round-robin schedulers schedule 87% of the packets to the GPU.

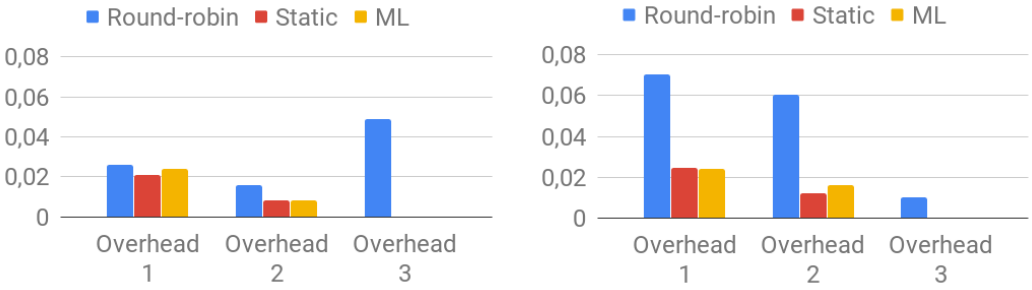


(a) Configuration 1

(b) Configuration 2

Figure 5.13: Packets successfully reordered during medium input test

The round-robin schedulers successfully reordered less than 10% of the packets, as shown in Figure 5.13. The C1 round-robin scheduler performed slightly better with 8.5% successful reorders while the C2 round-robin only had 6.5% successful reorders. The static and machine learning schedulers successfully reordered 100% of the packets.



(a) Configuration 1

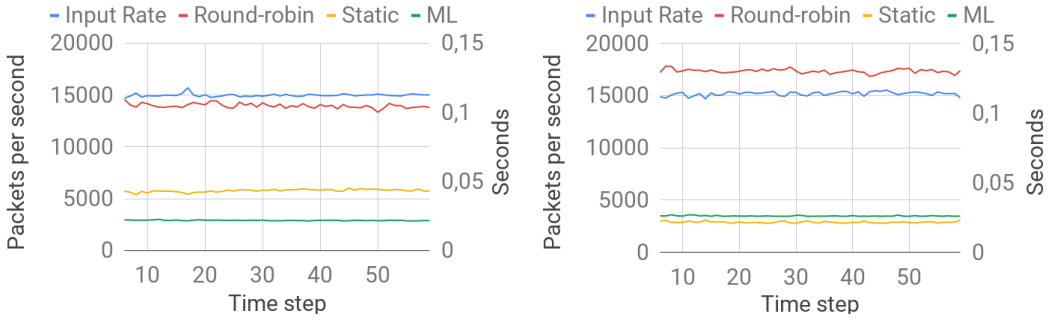
(b) Configuration 2

Figure 5.14: Overhead during medium input test

The C1 static scheduler has the lowest overhead for the first step which is shown as overhead 1. For overhead 2, the C2 machine learning scheduler showed the lowest overhead and neither of the static and machine learning schedulers had any significant overhead for overhead 3. The C2 round-robin scheduler clearly has the total highest overhead, as shown in Figure 5.14.

5.4 Stress Test

The stress test suit showcases how the schedulers behave during high traffic rates.

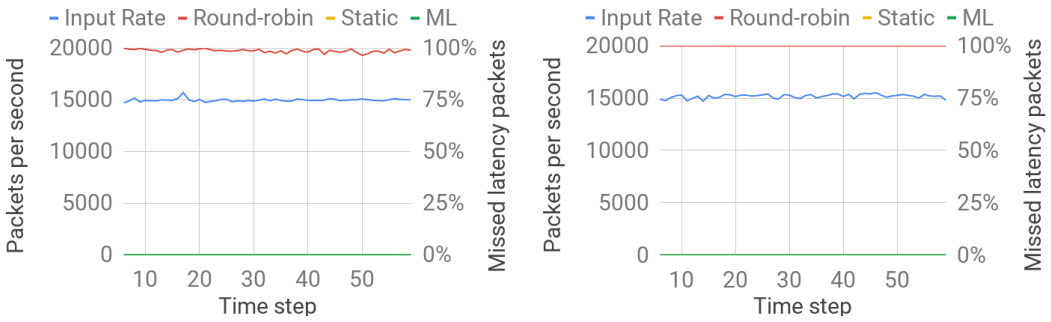


(a) Configuration 1

(b) Configuration 2

Figure 5.15: Stress test with continuous average latencies

As shown in Figure 5.15, the C2 round-robin scheduler has the highest latency for the stress test and the C1 machine learning scheduler has the lowest latency. The C2 machine learning scheduler and C2 static scheduler have slightly higher latency.

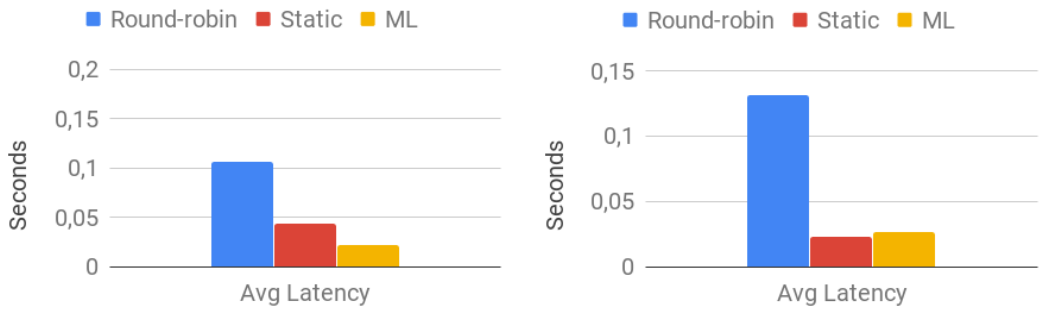


(a) Configuration 1

(b) Configuration 2

Figure 5.16: Missed latency-sensitive packets during stress test

As shown in Figure 5.16, the static and machine learning schedulers can handle the latency sensitive packets in a way that makes sure that they are processed in time. The round-robin schedulers do not prioritize latency-sensitive packets and therefore all or almost all of the packets miss their latency target.

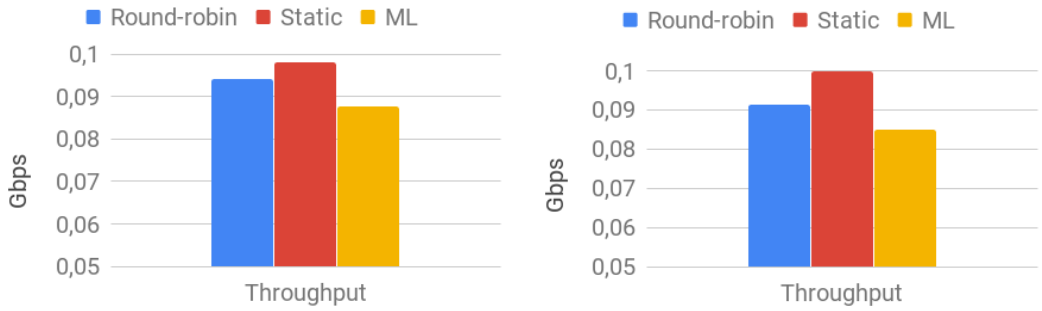


(a) Configuration 1

(b) Configuration 2

Figure 5.17: Average latency during stress test

Figure 5.17 show that the C1 machine learning scheduler and C2 static scheduler have the lowest and almost the exact same latency for the stress test. The C2 has a slightly higher average latency and the C1 static schedulers average latency is close to 0.05 seconds.

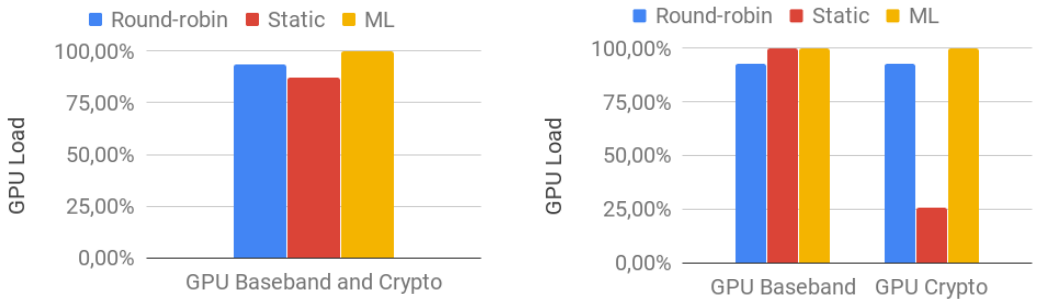


(a) Configuration 1

(b) Configuration 2

Figure 5.18: Average throughput during heavy input test

The static schedulers have the highest throughput for the stress test where the C2 static scheduler has an average throughput of 0.1 Gbps and the C1 have an average of 0.098 Gbps. The C2 machine learning scheduler has a slightly higher throughput compared to the C1 machine learning scheduler. The C1 round-robin has the second highest throughput for the stress test.

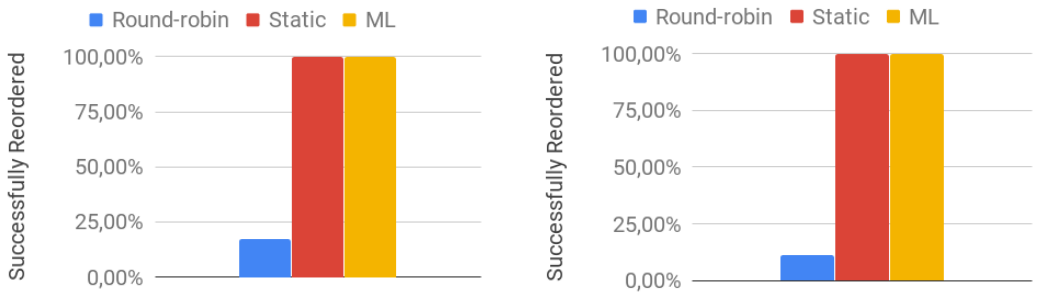


(a) Configuration 1

(b) Configuration 2

Figure 5.19: Average GPU load during heavy input test

Figure 5.19 shows that most of schedulers schedule between 87% and 100% of the packets to the GPU. The C2 static scheduler differentiates to the rest by scheduling 75% of the packet to be encrypted by the CPU workers and only 25% of the packets on the GPU while doing 100% of the LDPC decoding on the GPU.

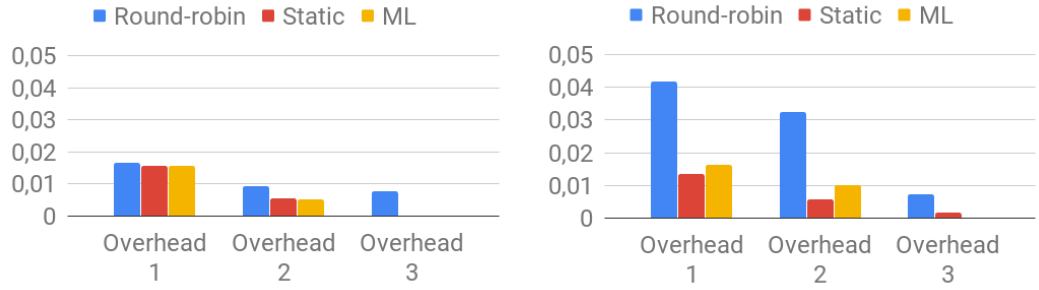


(a) Configuration 1

(b) Configuration 2

Figure 5.20: Packets successfully reordered during heavy input test

The static and machine learning schedulers successfully reordered all the packets during the stress test, as shown in Figure 5.20. The round-robin schedulers did not perform as well and the C1 scheduler successfully reordered 17% and the C2 scheduler successfully reordered 11% of the packets.



(a) Configuration 1

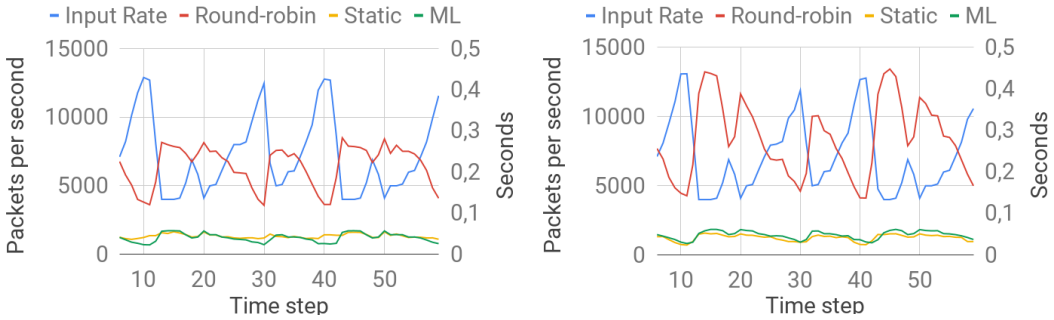
(b) Configuration 2

Figure 5.21: Overhead during heavy input test

It is important to notice the overhead 3 presented in Figure 5.21 for the C2 static scheduler. The other static scheduler and machine learning schedulers show no significant overhead for overhead 3. Otherwise, the static and machine learning schedulers show similar results.

5.5 Standard Test

The standard test suite showcases the resulting behavior of the scheduler implementations in a scenario simulating varying traffic rates.

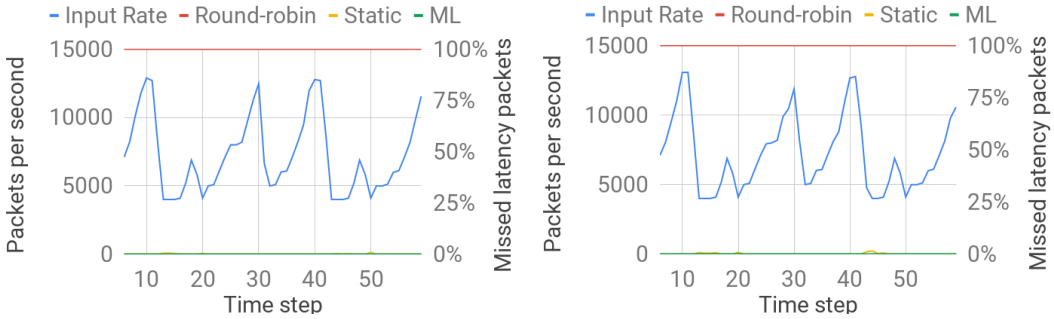


(a) Configuration 1

(b) Configuration 2

Figure 5.22: Standard input test with continuous average latencies

The static and machine learning schedulers maintain a lower average latency throughout the entire trace. They are also significantly better at responding to the varying conditions, having a more stable latency. The round-robin latency peaks are also closely mirrored to the varying conditions, which can also be identified in the static scheduler. This behavior is most likely a result of the GPU batch size aggregation, where segments of low or no input result in relatively high packet latencies. The latencies get lower when the input rates get higher and vice versa for when the input rates get lower.

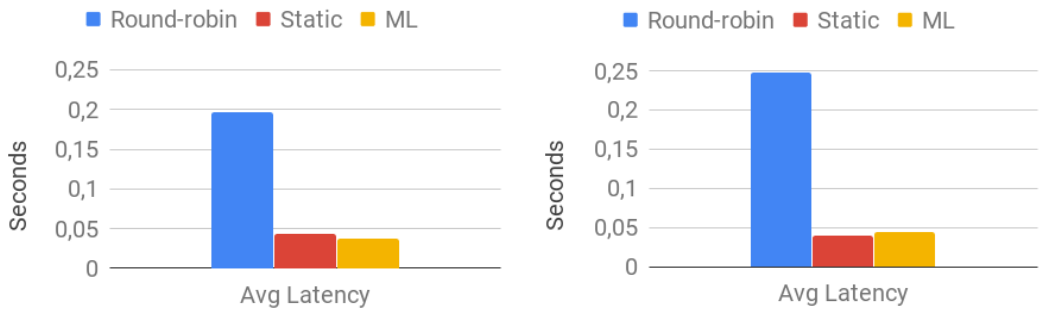


(a) Configuration 1

(b) Configuration 2

Figure 5.23: Missed latency-sensitive packets during standard input test

The round-scheduler does not handle latency sensitive packets as well as the static and machine learning schedulers as shown in Figure 5.23. It is possible to see that the C1 static scheduler miss 1% of the latency sensitive packets at time step 50 and that the C2 static scheduler miss 1% at time step 20, 43 and 44. These are time steps where the input drops from higher to lower input rates which is hard for the static schedulers to handle.

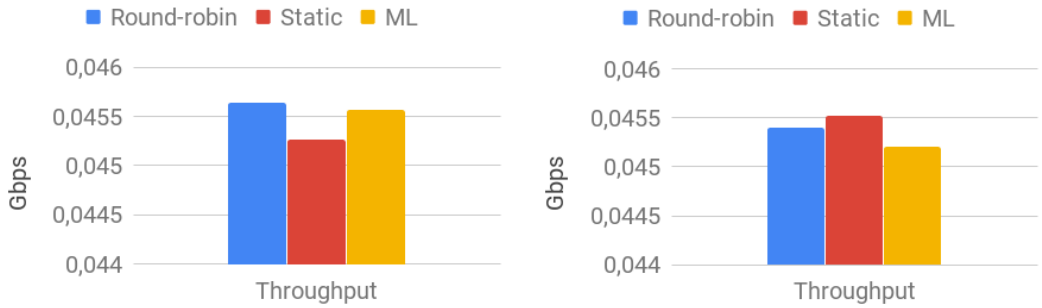


(a) Configuration 1

(b) Configuration 2

Figure 5.24: Average latency during standard input test

Figure 5.24 show that the C1 machine learning scheduler has the lowest average latency. The C2 machine learning and C1 and C2 static schedulers all have average latencies less than 0.05 seconds.

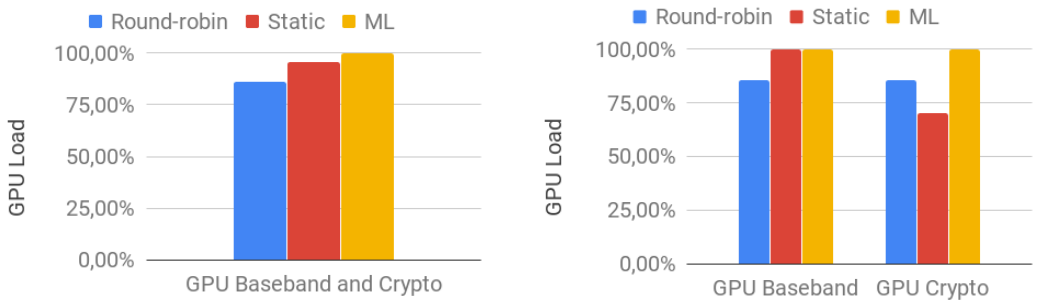


(a) Configuration 1

(b) Configuration 2

Figure 5.25: Average throughput during standard input test

The C1 round-robin scheduler has the highest throughput for the standard test with a throughput of 0.0456 Gbps. The C2 machine learning scheduler performed worst with a throughput of 0.0452 Gbps, the difference between the highest and lowest throughput is 0.97%.

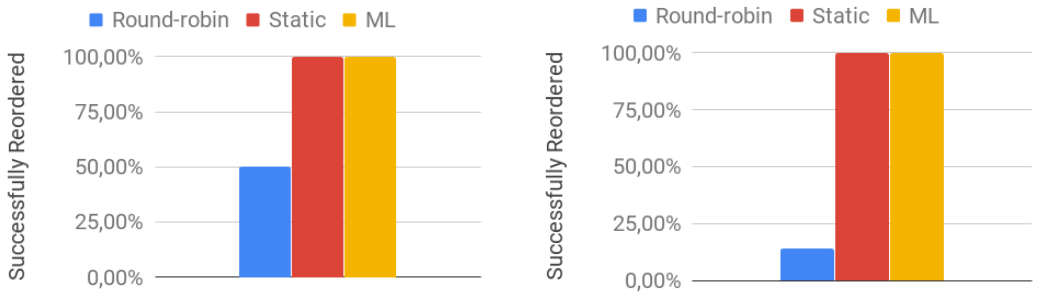


(a) Configuration 1

(b) Configuration 2

Figure 5.26: Average GPU load during standard input test

Figure 5.26 shows that machine learning schedulers distribute all the packets to the GPU. The C1 static scheduler distributes 96% to the GPU while the C2 static scheduler distributes all the packets to the baseband GPU worker and 70% to the crypto GPU worker.

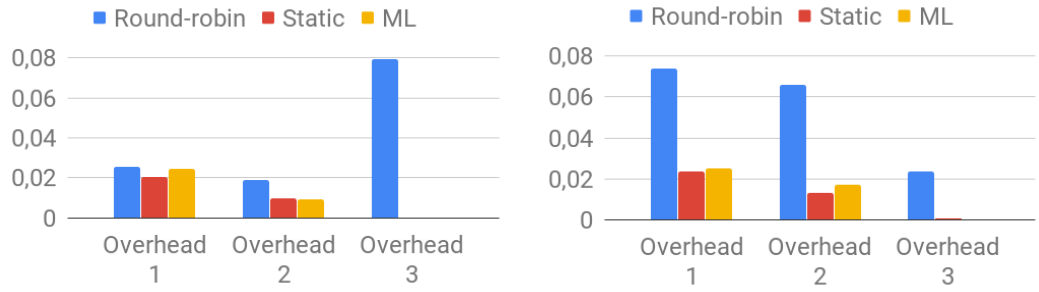


(a) Configuration 1

(b) Configuration 2

Figure 5.27: Packets successfully reordered during standard input test

The machine learning schedulers and static schedulers successfully reordered 100% of the packets. The round-robin schedulers performed worse with 50% successful reorder for the C1 scheduler and 14% successful reorder for the C2.



(a) Configuration 1

(b) Configuration 2

Figure 5.28: Overhead during standard input test

Figure 5.28 show that the static and machine learning schedulers have their largest overhead at the start, overhead 1. They do not have any significant overhead where the packets are reordered, overhead 3, because they schedule the packets on the same path through the application. This is not the case for the round-robin schedulers where reordering requirements are not taken into account.

5.6 Scheduler Summary

In this section summarized data across the entire test, runs are presented. Throughput, average latency, missed thresholds of latency sensitive packets and total overhead is shown for the different tests.

Table 5.1: Throughput, latency, latency-sensitive packets missed and total overhead for low input test

Scheduler	Throughput (Gbps)	Latency (ms)	Latency missed (%)	Total Overhead
C1 Round-robin	0.01890	0.30307	100	0.19863
C2 Round-robin	0.01887	0.48441	100	0.47554
C1 Static	0.01895	0.06829	20.13	0.05729
C2 Static	0.01895	0.06221	30.74	0.05737
C1 ML	0.01895	0.06913	21.44	0.06818
C2 ML	0.01895	0.08624	55.75	0.07517

The static and machine learning schedulers have the same throughput which makes it more interesting to compare their average latencies. The C2 static scheduler has the lowest latency while the C1 static scheduler has the lowest total overhead and also have the least missed latency-sensitive packets. The extra overhead is most noteworthy when comparing the C1 and C2 machine learning schedulers, where the difference in total overhead is 0.007 seconds, which can be the reason for the extra 34.31 percentage points in latency-sensitive packets missed.

Table 5.2: Throughput, latency, latency-sensitive packets missed and total overhead for medium input test

Scheduler	Throughput (Gbps)	Latency (ms)	Latency missed (%)	Total Overhead
C1 Round-robin	0.05048	0.19345	100	0.09097
C2 Round-robin	0.05050	0.22891	100	0.14115
C1 Static	0.05056	0.03764	0.0	0.02947
C2 Static	0.05056	0.03818	0.0	0.03659
C1 ML	0.05056	0.03510	0.0	0.03225
C2 ML	0.05056	0.04314	0.0	0.04009

The throughput is very similar for all the different schedulers, with a difference of 80 kbps between the highest and lowest average throughput. The difference between the C1 and C2 machine learning schedulers in latency is the 7.84 μs extra overhead that the C2 scheduler have. The C2 static scheduler has 0.00712 ms more total overhead but only have 0.54 μs higher average latency compared to the C1 static scheduler.

Table 5.3: Throughput, latency, latency-sensitive packets missed and total overhead for stress test

Scheduler	Throughput (Gbps)	Latency (ms)	Latency missed (%)	Total Overhead
C1 Round-robin	0.09420	0.10650	98.76	0.03357
C2 Round-robin	0.09122	0.13141	100	0.08165
C1 Static	0.09817	0.04324	0.0	0.02141
C2 Static	0.09982	0.02261	0.0	0.02094
C1 ML	0.08785	0.02227	0.0	0.02091
C2 ML	0.08501	0.02677	0.0	0.02627

The C1 and C2 static schedulers have higher average throughput than the machine learning and round-robin schedulers. During the stress test, the static schedulers benefit from offloading some packet to the CPU workers while the machine learning schedulers distribute all the packets to the GPU worker. Although the C1 machine learning scheduler has a lower throughput, it has the lowest latency and lowest total overhead.

Table 5.4: Throughput, latency, latency-sensitive packets missed and total overhead for standard test

Scheduler	Throughput (Gbps)	Latency (ms)	Latency missed (%)	Total Overhead
C1 Round-robin	0.04564	0.19705	100	0.12354
C2 Round-robin	0.04540	0.24718	100	0.16374
C1 Static	0.04527	0.04278	0.07	0.03008
C2 Static	0.04552	0.03920	0.11	0.03688
C1 ML	0.04556	0.03667	0.0	0.03411
C2 ML	0.04520	0.04461	0.0	0.04230

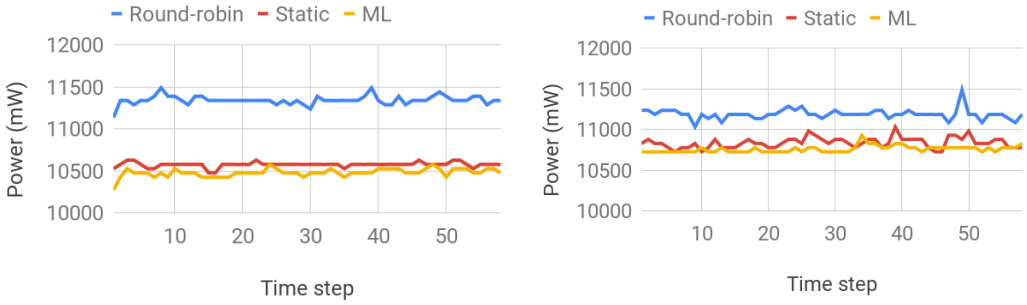
Without surprise, the round-robin schedulers perform very bad in latency thresholds achieved. They are however competitive in throughput. The static schedulers and ML schedulers have about equal results in terms of throughput but differentiate some in average latency.

5.7 Power Consumption

For the low input test and standard test, the power consumption is measured to better understand the schedulers different attributes. The reason for doing these measurements on the low input and standard test is because the schedulers showed similar results during these tests and to better differentiate which scheduler is the most beneficial to use, power consumption is also measured.

5.7.1 Low

The power consumption was measured for the GPU, CPU, SOC and DDR RAM during the low input test, the results from these measurements are shown below.



(a) Configuration 1

(b) Configuration 2

Figure 5.29: Energy used during low input test

Figure 5.29 show that the power consumption is steady through the whole test, which was expected since the input rate is steady during the test. It is also shown that the C1 static and ML schedulers uses less power than their C2 counterparts and that the C1 ML scheduler uses the least amount of power and that the C1 round-robin scheduler uses the most.

Table 5.5: Power distribution between GPU, CPU, SOC and DDR RAM for low input test

Scheduler	GPU (mW)	CPU (mW)	SOC (mW)	DDR RAM (mW)
C1 Round-robin	918	7366	2757	305
C2 Round-robin	918	7209	2758	304
C1 Static	1064	6752	2606	153
C2 Static	1072	6967	2647	153
C1 ML	1072	6648	2606	153
C2 ML	1072	6925	2609	153

Table 5.5 show that the round-robin schedulers distribute more power to the CPU than the other schedulers, which is expected since it distribute more packets to the CPU than the other schedulers. The difference in power distribution between the C1 and C2 static and ML schedulers is that the C2 schedulers distribute more power to the CPU. This is probably due to the extra decision point which the C2 schedulers has which requires extra power. As shown in Figure 5.5, the packet distribution is similar between the static and ML schedulers, which indicates that the difference should be the extra decision point.

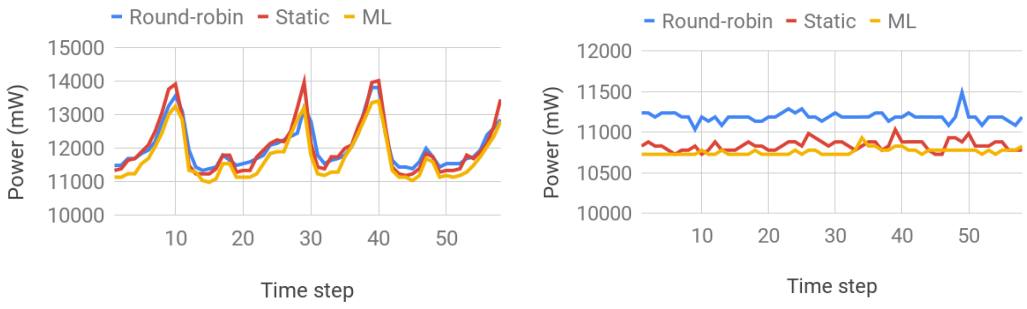
Table 5.6: Average power used during low input test

Scheduler	Average power (mW)
C1 Round-robin	11344
C2 Round-robin	11190
C1 Static	10575
C2 Static	10838
C1 ML	10479
C2 ML	10759

When the power distribution is put together, it is clear that the C1 static and ML schedulers uses less power than the their C2 counterparts, as shown in Table 5.6.

5.7.2 Standard

The power consumption was measured for the GPU, CPU, SOC and DDR RAM during the standard test, the results from these measurements are shown below.



(a) Configuration 1

(b) Configuration 2

Figure 5.30: Energy used during standard test

Figure 5.30 show that when the input rate peaks, the static schedulers consumes most power compared to the other schedulers. The ML schedulers consumes the least amount of power and have lower peaks than the other schedulers.

Table 5.7: Power distribution between GPU, CPU, SOC and DDR RAM for standard input test

Scheduler	GPU (mW)	CPU (mW)	SOC (mW)	DDR RAM (mW)
C1 Round-robin	1584	7329	2808	317
C2 Round-robin	1592	7365	2818	323
C1 Static	1741	7194	2843	263
C2 Static	1773	7184	2804	251
C1 ML	1782	6916	2818	231
C2 ML	1761	6914	2790	232

Similar to the power distribution results showed for the low input test 5.5, Table 5.7 show that the round-robin schedulers distributes less power to the GPU and more to the CPU compared to the other schedulers. The ML schedulers distributes all the packets to the GPU and while the static schedulers distribute most packets to the GPU, it also offload packets to the CPU, especially during the input rate peaks. This is why the static schedulers distribute more power to the CPU compared to the ML schedulers.

Table 5.8: Average power used during standard test

Scheduler	Average power (mW)
C1 Round-robin	12038
C2 Round-robin	12098
C1 Static	12040
C2 Static	12012
C1 ML	11747
C2 ML	11696

Table 5.8 show that the C2 ML scheduler consumes the least amount of energy during the standard test and that the C2 round-robin consumes the most. The ML schedulers clearly uses the least amount of power and uses approximately 0.3 W less than the static schedulers on average for the standard test.

6 Discussion

This chapter is divided into three sections, Section 6.1 discusses the results presented in Chapter 5. The second section, Section 6.2, discusses the methodology used and why they where chosen. The final section, Section 6.3, discusses the work in a wider context, the benefits and what conclusions that can be drawn from the this thesis.

6.1 Results

In this section, the results of the low input, medium input, stress and standard test and how the different schedulers handled them are analyzed and discussed.

6.1.1 Low Input Test

The low input tests highlighted the issues surrounding GPU buffer aggregation and the importance of a latency sensitive solution. While the batch sizes were dynamic to a certain extent, it is definitely an area of the scheduler which would require improvement. Comparatively between schedulers, C1 performed better than C2 for ML and worse for static.

The ML difference is not surprising, since both algorithms only utilized the GPU. This means the added overhead of the task scheduling C2 solution provides no benefit but only extra buffer steps. This is especially highlighted in latency sensitive packets missed, where that extra overhead tips over to a miss for over 50% compared to $\sim$25% of C1.

The static on the other hand utilizes this extra of C2 and can therefore slightly improve the average latency compared to C1.

The very high round-robin latencies is mainly due to enqueueing large packets on the CPU for baseband processing, but also a indirect result of the earlier mentioned GPU batch aggregation issues. Since the GPU already suffers from the low input rate, the CPU dispatching reduces the GPU packet rate even further.

6.1.2 Medium Input Test

The medium test had the same characteristics as low for the C1 and C2 versions of ML with a slight loss for ML in C2 when it came to average latency. The difference of 8.04 ms in average latency is approximately the same as the difference in total overhead, 7.84 ms. As can be seen in Figure 5.14, the biggest difference in the overhead is overhead 2 which also is expected to be bigger for the C2 ML scheduler compared to the C1 ML.

Both ML and static also ensures no latency sensitive packet misses for either C1 or C2, meaning the higher input rate is enough to keep the GPU batches flowing at a sufficient rate.

It is also interesting to note that during these rates the C2 static implementation starts utilizing the CPU to a greater extent without losing out on average latency compared to C1. This means the benefit of the task scheduling equals out the added overhead for the medium test.

As for the low input test, the throughput is still on a manageable level for all implementations and does therefore not stand out.

6.1.3 Stress Test

Stress testing the scheduler provides, for the context of a mostly GPU dominant performance as in our case, probably the most clear results in how well the schedulers can utilize the platforms without losing crucial performance. The stress test also reveals a notable difference in average latency for the two configurations, C1 and C2, of the Static scheduler where C2 is about twice as good. This is further strengthened in throughput, where Static performance is slightly better in C2.

This test is also clearly showing the gains of actually utilizing the entire platform well, increasing throughput of the Static compared to ML for both C1 and C2. For C2, this is also done while maintaining a lower average latency than the all out GPU processing of the ML implementation. This is where the second step of C2 shines, since the CPU can be utilized for crypto without throttling the baseband processing on the GPU, which remains the bottleneck.

For the ML configurations the results in terms of throughput is pretty much equal, with C1 very slightly ahead. The total overhead difference remains approximately the same as the other tests, with $\sim 0,006$ lower overhead for C1.

At the same time, as for the low and medium tests, the Round-robin performs a lot worse in the C2 context. This is because the added step comes with extra local buffers for in between packet storage together with an extra ring buffer, all this for each worker. An important failure of the Round-robin is that in all scenarios it sends far too much to the CPU. Even during low input rates the slow baseband processing of the CPU with the relative high CPU dispatching from Round-robin means it is constantly maxed out. Since this results in full buffers, there are a lot of packets, on top of using the low CPU path, also having to wait for other packets being slowly processed by the CPU. This is a major cause for the high latencies, and in C2, these buffers are doubled with the extra enqueue step which results in even more packets having to deal with this waiting. The quite extreme differences is partly a consequence of the schedulers maintaining too large 128 slot ring buffers towards the CPU, resulting in unnecessarily large amount of packets wasting time in buffers. These sizes were quite naively chosen from existing DPDK example applications, but they can most certainly be made smaller without losing any performance. To ensure this, a few extra tests were made where all input ring buffer sizes were halved to 64 slots. Since the sizes of ring buffers are required to a be a power of two it is the smallest reasonable size to allow burst sizes of 32 packets with some headroom to ensure the ring buffers are not throttling the scheduler. These results are shown below in Table 6.1.

Table 6.1: Ring buffer size decrease test

Test	Ring buffer size	Throughput (Gbps)	Average latency (s)
Heavy	64	0.09469	0.1074
Heavy	128	0.09494	0.1268
Low	64	0.01891	0.2615
Low	128	0.01886	0.5072

All tests were made for C2 Round-robin, since the overhead impact of the buffers affected that scheduler most. The reduction in size and consequently waiting times resulted in an $\sim 49.5\%$ decrease in average latency, showing the impact of the undesirable CPU usage. Even for the heavy traffic stress test the average latency could be reduced by $\sim 13.3\%$. As previously mentioned, any smaller buffers were not desired since this can cause undesirable idle times of the actual processing due to a reactionary polling rather than a proactive buffer, always having packets ready for the next batch.

These extra buffers are also partly the reason, together with the unavoidable extra moving of pointers, for the latency disparity between C1 and C2 for ML. They are not as extreme however since the GPU works a lot faster, meaning even for full buffers the packets do not have to wait as long. The larger batches of the GPU also means the larger buffers are better suited for the GPU worker.

6.1.4 Standard Test

As mentioned in the discussion of the lower test, the aggregation of batches was a major issue during low input rates. This is further highlighted by the standard test suites. The pattern for all scheduler implementations, though the Round-robin is clearly dominant, shows that average packet latencies increases during low rates and decreases during higher input rates.

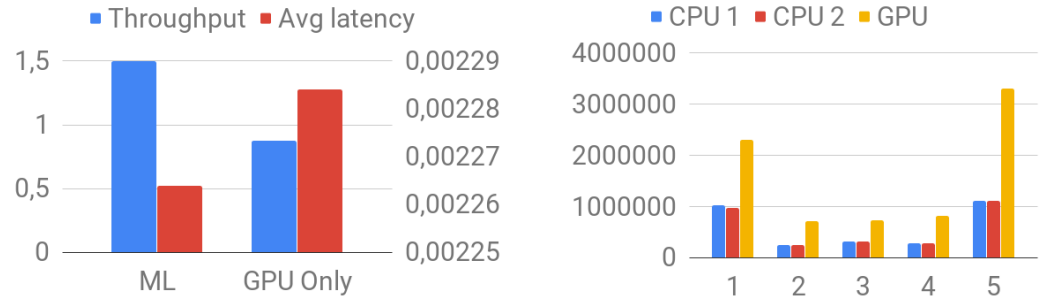
Despite this, the results follow the same relative pattern as the stress test. For C1 the ML actually outperforms Static in average latency, throughput and missed latency requirements. For C2 however the opposite applies, where Static is better in all instances except missed latency requirements, which occurs during quick and large input rate drops.

6.1.5 Scheduler Summary

To summarize, the C2 Static scheduler had the strongest performance over most metrics, since it best could utilize the systems resources to maximize the throughput of the baseband bottleneck. Sadly the ML implementations could not compete for the high load where the CPU offloading became beneficial. At the same time it highlighted the dominance of the GPU further, and since it did provide lower latencies across most tests, together with zero latency sensitive misses except for low, the result is not surprising for these scenarios. It was clear how penalizing the effect of more overhead for C2 became during bad decisions. Since one of the reasons described was the unnecessarily large buffers, a final test with Round-robin was made with smaller buffers of 64 slots, which should be more appropiate for the maximum bursts of 32 packets with some headroom.

The reason why the ML could not see the benefit of CPU dispatching even for heavy traffic goes back to the discussed issue about throughput mentioned in Chapter 4 when discussing the reward function. While the latency coupled rewards were easy to tie to a certain packet and action, the throughput is not. This means actions that an action for dispatching a packet to the CPU and increasing throughput, will also increase the throughput for every other action taken in within approximately the same time. Basically the GPU actions benefit aswell, making it hard to distinguish the a sole benefactor since all actions recieve credit.

The issue surrounding this could perhaps be atleast partially handled by introducing a different training session. Steering the training towards different scenarios, such as pure GPU processing, and letting the algorithm learn during training based on so far aggregated data. During development this was avoided due to the aim of adaptive behaviour and the risk of over-fitting. Furthermore, due to the limiting throughput bottleneck of the baseband, the possible increase as seen from summary results were at most 17.5% when comparing C2 Static and C2 ML. Because of these issues, we must admit that the algorithm is not advanced enough to deal with two processing tasks effect on each other and their contention on the same resources.



(a) Throughput (Gbps) and Latency Comparison (s)

(b) Packet Distribution

Figure 6.1: ML comparison for cryptography processing

However, just to verify the feasibility of the algorithm for a simpler scenario with a more interesting trade-off between CPU and GPU from a competetive perspective the algorithm was applied on a scheduler only running crypto processing. The results are shown in Figure 6.1a. The test ran was a stress test since this scenario provides the best beneficial consequences of full resource utilization. It is compared to a crypto-only scheduler which performs all processing in the GPU, same as the other ML schedulers running both tasks. As seen, without the added complexity of the baseband, the algorithm selects a distribution which almost doubles throughput performance while also decreasing average latency. The distribution can be seen in Figure 6.1b, where the x-axis numbers correspond to packet types. This further enforces the above thesis that the algorithm requires a more complex implementation regarding the interaction between tasks.

6.1.6 Power Consumption

The results from the power evaluation of schedulers in comparable throughput conditions further reinforced the benefit of GPU usage. Over the two configurations a slight increase in CPU power usage could be seen for C2, which can be expected due to the added overhead work. The Round-robin schedulers did however have opposite relations, but these are harder to draw conclusions from since they are not as predictable in their distribution. This means the difference is probably not due to configurations, but rather one configurations averaging a worse distribution in terms of what types of packets are sent to which platform.

The more interesting comparison is between the pure GPU-dispatching ML scheduler and the CPU-utilizing Round-robin. For the low test, the approximately 30% of total packets are sent to the CPU, seen in Figure 5.5, results in an overall 865 mW and 431 mW increase for C1 and C2 respectively. While the heavier GPU usage of the ML scheduler is reflected in the GPU power consumption, all other parts of the system during CPU processing resulting in a greater efficiency loss. The pattern is also the same for standard tests, except for the SOC power values, which seems more stable during these tests. This is probably due to parts of the SOC being idle to a greater extent during the low tests utilizing the GPU only.

It can also be worth highlighting the comparatively high CPU power consumption even during full GPU processing is simply because, on top of the actual scheduling loop, the CPU workers still run and poll for packets, even when none are distributed to them. Also, which is briefly discussed and considered further down in the method section, the test application also operates on this CPU.

Table 6.2: CPU worker expansion for C1 running the low input test

Scheduler	CPU Workers	GPU (mW)	CPU (mW)	SOC (mW)	DDR RAM (mW)
Round-robin	1	920	6411	2607	153
Round-robin	2	918	7366	2757	305
Round-robin	3	918	8396	2757	306
Round-robin	4	855	9339	2754	306
ML	1	1073	5608	2607	153
ML	2	1072	6648	2606	153
ML	3	1072	7814	2604	153
ML	4	1071	8743	2639	153

To provide some extra insight in the impact of the CPU workers on power consumption and the cost of further utilization of the CPU, a couple of extra tests were performed and using varying numbers of CPU workers. These were only made for the low test using the C1 configurations of the ML and Round-robin schedulers. Since the patterns between C1 and C2 were similar, the ML and Round-robin most distinguishable and low provides the most equal throughput the exploration were considered sufficient to limit to these scenarios. Table 6.2 shows the new power consumption values measured for the added worker test together with the values from the original tests, retrieved from the result section. The data shows a significant increase in power consumption for the CPU with the added workers, where the especially interesting aspect is the approximately 1000 mW surge for the ML scheduler per worker. This is despite not actually doing any CPU processing, but just the extra idle worker loops.

Another result from this is that the rise in power consumption is the same for the Round-robin, meaning the difference in CPU consumption between Round-robin and ML does not grow above the original difference of approximately 600 mW. Since the packets handled by the CPU does not double when the amount of workers do, rather increase by $\sim$ 30%, the gap was not expected to grow at a 600 mW rate but it was still assumed the processing would yield greater power consumption due to the original difference. Differences from results with Static involved shows that the minor CPU dispatching done by the Static scheduler results in a slight increase of power consumption compared to ML, while significantly lower than Round-robin. With this in mind it is hard to pinpoint the reason why the samples maintain a a steady increase, rather than reflecting the increased CPU processing in growing CPU power consumption. One important difference between changing worker count for the Round-robin and switching between Round-robin and Static however is that the Round-robin always maxes out the CPU for each worker count. This means the same number of packets are dispatched to each CPU worker for both low and heavy load tests. It therefore appears, that the gap between the full GPU utilization of the ML scheduler to the full CPU utilization of the Round-robin only grows until this full CPU utilization is achieved. After that point, the increase in worker count comes with the same cost whether or not any processing is performed.

The values for SOC and DDR RAM also reveals a threshold between Round-robin runs of one and two CPU workers when a sufficient number of packets are dispatched to the CPU to require extra resources from these system components to kick in.

With this price in mind, a power efficient setup, in the context of the performance presented in this thesis, should only set up enough CPU workers to keep up with the baseband processing of the GPU. Any unnecessary CPU worker comes at a significant power cost. On a positive note, the

increase is actually the same for the Round-robin which actually utilizes the CPU. This indicate that the idle worker loops perhaps can be configured to allow a dormant state, for example during low traffic. Such a solution could save a significant amount of energy while maintaining the resource.

6.2 Method

In this section, the process of researching and developing the scheduler and its components is discussed. The chapter is split into three main sections which are considered the most important parts in terms of effort, time and relevance.

6.2.1 Poll Mode Drivers

The drivers performing the processing operations turned out to be a major effort, especially since it was initially believed that existing drivers would be utilized, at least for the CPU. Instead of using existing and already optimized drivers, we had to find and integrate open-source implementations of the desired baseband and encryption processing.

The encryption driver was the first one to be set up. While there existed encryption drivers for AES-CBC encryption on ARM64 architectures, it would be strange to implement such a sequentially bound algorithm on the GPU. Therefore time and effort went into finding a suitable library for AES-CTR, both for CPU and GPU, and integrating these through two new PMDs. Sadly these implementations are not state of the art implementations, and the CPU code especially was far from the performance of the assembly optimized CPU AES-CBC driver. However, optimizing these parts of the code felt out of scope. So as long as the relative time felt reasonable we settled with putting some effort into the memory transfer and setup between CPU and GPU.

AES-CTR is more well known than SNOW 3G and ZUC and it was, therefore, easier to find good open-source solutions. AES-CTR, SNOW 3G and ZUC are all stream-ciphers and uses similar techniques to encrypt data, it would therefore probably not make a big difference in throughput if we would have chosen one of the other methods. This is something that would be interesting to investigate in the future though.

The baseband driver took a considerable amount of time to set up and get working. Since there were only two existing baseband drivers, the null driver, and turbo code driver, there were not many guidelines on how to create a good baseband driver. The existing turbo code was written for an Intel CPU and since the hardware we wanted to use were using an ARM CPU, we could not use that code either. LDPC was mainly chosen because it will be used in the 5G networks, because it is parallelizable and therefore suited to be utilized on a GPU and because previous research has also found that LDPC is close the Shannon Limit. Other examples of error correction codes that have been accepted in the 5G standardization are turbo code and polar code which we also considered but because of reasons stated above and in Section 2.3, LDPC became our method of choice.

Similar to the AES-CTR implementation, the LDPC implementation is far from state of the art and could probably be optimized significantly. We also focused on the decoding of the LDPC code since that is what uses the most computational resources and is most interesting. The LDPC decoding CPU implementation turned out to be a significant bottleneck for the whole application. That it would perform worse than the GPU implementation was expected, but not that it would perform 11 to 35 times worse. We have found previous research [46] where an ARM CPU implementation achieve a throughput of 50 to 100 Mbps which would have provided a more competitive implementation against the GPU. However, due to not finding a good open-source solution and time constraints which prevented us from trying to improve our own solution further, we had to make due with the solution we had at our hands.

The GPU implementation also turned out to be the bottleneck for both the latency and throughput of the application and the maximum throughput of 130 Mbps for IMIX is lower than the throughput of our cryptography implementation on one CPU core, which is 308.9 Mbps.

6.2.2 Scheduler

While a baseline is often some current implementation or previous comparative study, this project lacked that. Therefore, in order to get some sense of improvement and gain in the scheduler decisions, we created a simple and primitive baseline for relative comparison. The round-robin scheduler was chosen for this purpose. While one could argue for a somewhat more clever baseline, we think the simplistic nature of the round-robin scheduling gave a good understanding of the potential pits and consequences of some decisions.

The reinforcement learning method was chosen because it suited our problem the best. It is a common practice for similar tasks where the aim is to explore potential gains and consequences, as for our purpose of evaluating the static scheduler and provide insight. However, developing such an algorithm is not trivial and the complexity of the approach can be unlimited. For our limited time, the goal to capture the important and intuitive aspects of a schedulers performance seemed reasonable. Sadly, the results of the machine learning was perhaps a bit too straight-forward. This was partly due to the issues, from a more complex scheduling point of view, surrounding the baseband drivers and the actual dominant performance of the GPU, making the scheduling somewhat dull.

6.2.3 Test Application

Overall the test application worked well and was quite fast to develop. The main consideration is probably that it is a DPDK application. This meant we could utilize DPDK specific structures and functions to represent and modify packets. However, a real scenario would have to allocate these structs and designate them to packets upon entering the RX port of the scheduler. But since our interest is the measurement of latencies inside the scheduler and the consequences of processing destination, the missing allocation step has little or no impact on our relative results.

Running the test application on the same platform also meant the work it performed had an impact on the power evaluation. Using the software provided by the Jetson Xavier it did not seem to be a way to isolate the measurements to certain cores. However, since the same measurements applied to each scenario the relative difference should at least be negligible. As described in Chapter 4, the power consumption samples did also add some extra overhead but as mentioned above this should apply to all tests and therefore not matter in the relative comparisons.

6.3 The Work in a Wider Context

5G will result in higher amount of data and internet throughput which will undoubtedly affect society in many ways. Smarter cities and self-driving cars are just some of the possible outcomes from the change to 5G, but it will also push forward innovation in sectors such as agriculture and health care. Wearable devices that monitor your health and sends direct feedback to the hospital when you get sick is a possible innovation that might be more common in the future. With so many devices connected, latency and throughput will be critical factors for it all to work together. By continuing researching how to lower latency, increasing throughput and utilizing the available systems in a power-efficient way, we take steps closer to a society where everything can be connected.

Through advancements in the processing capabilities surrounding data traffic such as encryption and baseband, 5G and the internet speed it promises can be ensured while maintaining important responsibilities towards users. These responsibilities correspond to maintain a safe environment where integrity and exposure of data is controlled which must not be neglected at the cost of speed.

7 Conclusions and Future Work

The thesis set out to investigate possibilities surrounding scheduling of packet processing tasks utilizing DPDK and, at the time of writing, the newly released integrated CPU/GPU Jetson Xavier from Nvidia. By considering the expected behavior of this heterogeneous system performing the processing tasks AES-CTR and LDPC, a static scheduler based on theoretical properties was developed. To validate, and explore potential benefits in a more adaptive scheduler with the potential of conforming to new environments, a machine learning scheduler was also created. A final round-robin scheduler was also used as a relative base-line and to show the potential benefit of smarter scheduling as well as the penalty of bad decisions.

Compared to the baseline round-robin both the static and machine learning approaches had no issue outperforming the baseline in most scenarios with up to 6.31 and 5.54 times lower average latencies for static and ML respectively. The difference was especially large in fulfilling sensitive latency requirements, where the Round-robin almost missed 100% for every test and configuration, while both ML and static could maintain around 0% loss for every scenario except the very low rates.

On top of these schedulers, two different configurations, *C1* and *C2*, of the DPDK implementation were also evaluated to provide an understanding of the impact of a more flexible, but more overhead, task scheduling compared to a more static approach with less overhead.

Based on these configurations, the results were that a more flexible scheduler with the added overhead of steps and buffers could be heavily punished if not utilized well, but as shown by the C2 version of the static scheduler the flexibility produced the best result in balancing throughput, average latency, and latency requirements. More importantly, the results highlighted several important aspects of scheduling on a heterogeneous system utilizing GPU acceleration. One of the main challenges is to effectively aggregate batches during low traffic rates to avoid a pattern of high latencies as the input rate decreases.

In scenarios where the throughput performance was similar between the schedulers, it became clear that the static and ML schedulers were also more power-efficient than the round-robin schedulers with up to 769 mW and 865 mW less power used for the static and ML schedulers respectively. This further enforces the benefit of using the GPU as much as possible in the context of this study with similar processing tasks.

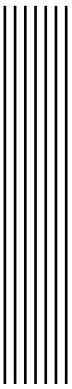
7.1 Future Work

The LDPC and AES-CTR implementations, which have been discussed in Section 6.2, could use some major improvements and optimization. Especially the CPU implementations would have been interesting to improve to make them more realistically comparable against the GPU implementations. A more competitive CPU implementation would have affected the whole system and the schedulers together with the applicability of the results in a state of the art environment.

While the Jetson Xavier integrated CPU/GPU and its properties are mentioned several times, any comparable analysis was unfortunately never made. To clearly get a picture of the benefit provided by the faster shared physical memory it would be beneficial to implement the scheduler on a system consisting of the more traditional CPU and discrete GPU. The power consumption evaluation also provided insight in the system components impact on the power efficiency, however the scheduling algorithms in this thesis were only evaluated based on this property. Future setups of resources and scheduling should also consider this in the decision process of configuring the system and dispatching data packets.

The machine learning algorithm introduced was concluded to be too simple, in what information and state it maintained, for the environment in which it operated. While it proved more promising in a single processing scheduling setup, it would be interesting to further examine the possibilities and requirements for a stronger performance in the multi-processing and resource contending implementation evaluated in this thesis.

It would also be interesting to implement a technique similar to the one described by Maghazeh et al. [29] which is described in Section 3.8. While our system had a dynamic batch size, it was not as good as the one described by Maghazeh et al. They used a CPU thread to monitor the input rate to make the batch sizes more adaptable for the changes in input rate. A more dynamic batch aggregation would help our schedulers to handle low input rates while still achieving high throughput. Their use of a persistent kernel, and therefore avoiding the overhead of launching new GPU kernels, would also be interesting to implement into the system.



Bibliography

[1] I. Parvez, A. Rahmati, I. Guvenc, A. I. Sarwat, and H. Dai. "A Survey on Low Latency Towards 5G: RAN, Core Network and Caching Solutions". In: *IEEE Communications Surveys Tutorials* 20.4 (2018), pp. 3098–3130. ISSN: 1553-877X. DOI: 10.1109/COMST.2018.2841349.

[2] A. Maghazeh, U. D. Bordoloi, P. Eles, and Z. Peng. "General purpose computing on low-power embedded GPUs: Has it come of age?" In: *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. July 2013, pp. 1–10. DOI: 10.1109/SAMOS.2013.6621099.

[3] Dustin Franklin. *NVIDIA Jetson AGX Xavier Delivers 32 TeraOps for New Era of AI in Robotics*. 2018. URL: https://devblogs.nvidia.com/nvidia-jetson-agx-xavier-32-teraops-ai-robotics/ (visited on 02/11/2019).

[4] DPDK Project. *About DPDK*. URL: https://www.dpdk.org/about/ (visited on 11/20/2018).

[5] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle. "A study of network stack latency for game servers". In: *2014 13th Annual Workshop on Network and Systems Support for Games*. Dec. 2014, pp. 1–6. DOI: 10.1109/NetGames.2014.7008960.

[6] C. Cirstea, R. Davidescu, and A. Gontean. "A reinforcement learning strategy for task scheduling of WSNs with mobile nodes". In: *2013 36th International Conference on Telecommunications and Signal Processing (TSP)*. July 2013, pp. 348–353. DOI: 10.1109/TSP.2013.6613950.

[7] Y. He, Z. Zhang, and Y. Zhang. "A Big Data Deep Reinforcement Learning Approach to Next Generation Green Wireless Networks". In: *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*. Dec. 2017, pp. 1–6. DOI: 10.1109/GLOCOM.2017.8254717.

[8] Evangelos Haleplidis, Kostas Pentikousis, Spyros Denazis, JH Salim, D Meyer, and O Koufopavlou. *Software-Defined Networking (SDN): Layers and Architecture Terminology*. RFC 7426. RFC Editor, Jan. 2015, pp. 1–35. URL: https://tools.ietf.org/html/rfc7426.

[9] Zongyao Li. "HPSRouter: A high performance software router based on DPDK". In: *2018 20th International Conference on Advanced Communication Technology (ICACT)*. IEEE. 2018, pp. 503–506. DOI: 10.23919/ICACT.2018.8323809.

[10] Prajwol Kumar Nakarmi Karl Norrman and Eva Fogelström. *5G security - enabling a trustworthy 5G system*. 2018. URL: https://www.ericsson.com/en/white-papers/5g-security---enabling-a-trustworthy-5g-system (visited on 03/25/2019).

[11] Federal Information Processing Standards Publications. *ADVANCED ENCRYPTION STANDARD (AES)*. Tech. rep. National Institute of Standards and Technology, Nov. 2001. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

[12] R. Ketata, L. Kriaa, L. A. Saidane, and G. Chalhoub. "Detailed analysis of the AES CTR mode parallel execution using OpenMP". In: *2016 International Conference on Performance Evaluation and Modeling in Wired and Wireless Networks (PEMWN)*. Nov. 2016, pp. 1–9. DOI: 10.1109/PEMWN.2016.7842901.

[13] Nhat-Phuong Tran, Myungho Lee, Sugwon Hong, and Seung-Jae Lee. "Parallel execution of AES-CTR algorithm using extended block size". In: *2011 14th IEEE International Conference on Computational Science and Engineering*. IEEE. 2011, pp. 191–198. DOI: 10.1109/CSE.2011.43.

[14] K Moon Todd. "Error correction coding: mathematical methods and algorithms. 2005 by John Wiley & Sons". In: ().

[15] Sijie Cheng. "Comparative Study on 5G Communication Channel Coding Technology". In: *3rd International Conference on Mechatronics Engineering and Information Technology (ICMEIT 2019)*. Atlantis Press. 2019. DOI: 10.2991/icmeit-19.2019.13.

[16] T. Richardson and S. Kudekar. "Design of Low-Density Parity Check Codes for 5G New Radio". In: *IEEE Communications Magazine* 56.3 (Mar. 2018), pp. 28–34. ISSN: 0163-6804. DOI: 10.1109/MCOM.2018.1700839.

[17] S. Shao, P. Hailes, T. Wang, J. Wu, R. G. Maunder, B. M. Al-Hashimi, and L. Hanzo. "Survey of Turbo, LDPC and Polar Decoder ASIC Implementations". In: *IEEE Communications Surveys Tutorials* (2019), pp. 1–1. ISSN: 1553-877X. DOI: 10.1109/COMST.2019.2893851.

[18] Lars Lundheim. "On Shannon and" Shannon's Formula"". In: *Telektronikk* 98.1 (2002), pp. 20–29. URL: http://www.cs.miami.edu/home/burt/learning/Csc524.142/LarsTelektronikk02.pdf.

[19] David J. C. MacKay. *Information Theory, Inference & Learning Algorithms*. New York, NY, USA: Cambridge University Press, 2002. ISBN: 0521642981.

[20] D.J.C. MacKay and R.M. Neal. "Near Shannon limit performance of low density parity check codes". In: *Electronic Letters* 32.18 (Aug. 1996). An optional note, pp. 1645–1646. DOI: 10.1049/el:19970362.

[21] J. Kim Y. Jung C. Chung and Y. Jungand. "7.7Gbps encoder design for IEEE 802.11n/ac QC-LDPC codes". In: *2012 International SoC Design Conference (ISOCC)*. Nov. 2012, pp. 215–218. DOI: 10.1109/ISOCC.2012.6407078.

[22] G. Wang, M. Wu, B. Yin, and J. R. Cavallaro. "High throughput low latency LDPC decoding on GPU for SDR systems". In: *2013 IEEE Global Conference on Signal and Information Processing*. Dec. 2013, pp. 1258–1261. DOI: 10.1109/GlobalSIP.2013.6737137.

[23] Sebastian Altmeyer, Sakthivel Manikandan Sundharam, and Nicolas Navet. *The case for FIFO real-time scheduling*. Tech. rep. University of Luxembourg, 2016. DOI: 10.13140/RG.2.1.4117.9924.

[24] Ellen L. Hahne. "Round-robin scheduling for max-min fairness in data networks". In: *IEEE Journal on Selected Areas in communications* 9.7 (1991), pp. 1024–1039. DOI: 10.1109/49.103550.

[25] P. Gupta and N. McKeown. "Algorithms for Packet Classification". In: *Netwrk. Mag. of Global Internetwkg.* 15.2 (Mar. 2001), pp. 24–32. ISSN: 0890-8044. DOI: 10.1109/65.912717.

[26] Seattle Internet Exchange. *SIX - Statistics Weekly*. 2019. URL: https://www.seattleix.net/statistics/#weekly (visited on 04/19/2019).

[27] AMS-IX. *AMS-IX Statistics - sFlow Statistics*. 2019. URL: https://stats.ams-ix.net/sflow/size.html (visited on 04/19/2019).

[28] William Stallings. *Computer organization and architecture : designing for performance*. Pearson Prentice Hall, 2010. ISBN: 9780136073734.

[29] "Latency-aware packet processing on CPU-GPU heterogeneous systems." In: *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC), Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE* (2017), p. 6. ISSN: 978-1-4503-4927-7. DOI: 10.1145/3061639.3062269.

[30] Janet Tseng, Ren Wang, James Tsai, Saikrishna Edupuganti, Alexander W Min, Shinae Woo, Stephen Junkins, and Tsung-Yuan Charlie Tai. "Exploiting integrated GPUs for network packet processing workloads". In: *2016 IEEE NetSoft Conference and Workshops (NetSoft)*. IEEE. 2016, pp. 161–165. DOI: 10.1109/NETSOFT.2016.7502464.

[31] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. "Addressing Shared Resource Contention in Multicore Processors via Scheduling". In: *SIGPLAN Not.* 45.3 (Mar. 2010), pp. 129–142. ISSN: 0362-1340. DOI: 10.1145/1735971.1736036.

[32] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018. ISBN: 0262193981.

[33] Abhishek Nandy and Manisha Biswas. *Reinforcement Learning: With Open AI, TensorFlow and Keras Using Python*. Apress, 2017. DOI: 10.1007/978-1-4842-3285-9.

[34] L. Zhao and Z. Chen. "Optimizing Quality of Experience of Free-Viewpoint Video Streaming with Markov Decision Process". In: *2018 IEEE International Conference on Communications (ICC)*. May 2018, pp. 1–6. DOI: 10.1109/ICC.2018.8422860.

[35] J. Tseng, R. Wang, J. Tsai, S. Edupuganti, A. W. Min, S. Woo, S. Junkins, and T. C. Tai. "Exploiting integrated GPUs for network packet processing workloads". In: *2016 IEEE NetSoft Conference and Workshops (NetSoft)*. June 2016, pp. 161–165. DOI: 10.1109/NETSOFT.2016.7502464.

[36] Uri Verner, Assaf Schuster, and Mark Silberstein. "Processing Data Streams with Hard Real-time Constraints on Heterogeneous Systems". In: *Proceedings of the International Conference on Supercomputing*. ICS '11. Tucson, Arizona, USA: ACM, 2011, pp. 120–129. ISBN: 978-1-4503-0102-2. DOI: 10.1145/1995896.1995915.

[37] H. Eom, P. S. Juste, R. Figueiredo, O. Tickoo, R. Illikkal, and R. Iyer. "Machine Learning-Based Runtime Scheduler for Mobile Offloading Framework". In: *2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*. Dec. 2013, pp. 17–25. DOI: 10.1109/UCC.2013.21.

[38] J. Li, X. Ma, K. Singh, M. Schulz, B. R. de Supinski, and S. A. McKee. "Machine learning based online performance prediction for runtime parallelization and task scheduling". In: *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. Apr. 2009, pp. 89–100. DOI: 10.1109/ISPASS.2009.4919641.

[39] J. Liu, Y. Mao, J. Zhang, and K. B. Letaief. "Delay-optimal computation task scheduling for mobile-edge computing systems". In: *2016 IEEE International Symposium on Information Theory (ISIT)*. July 2016, pp. 1451–1455. DOI: 10.1109/ISIT.2016.7541539.

[40] Y. R. Qu, H. H. Zhang, S. Zhou, and V. K. Prasanna. "Optimizing many-field packet classification on FPGA, multi-core general purpose processor, and GPU". In: *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. May 2015, pp. 87–98. DOI: 10.1109/ANCS.2015.7110123.

[41] Waqar Ali and Heechul Yun. “Protecting Real-Time GPU Applications on Integrated CPU-GPU SoC Platforms”. In: *arXiv preprint arXiv:1712.08738* (2017). URL: https://arxiv.org/pdf/1712.08738.pdf.

[42] Dustin Franklin. *NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge*. 2017. URL: https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/ (visited on 02/11/2019).

[43] W. Zhu, P. Li, B. Luo, H. Xu, and Y. Zhang. “Research and Implementation of High Performance Traffic Processing Based on Intel DPDK”. In: *2018 9th International Symposium on Parallel Architectures, Algorithms and Programming (PAAP)*. Dec. 2018, pp. 62–68. DOI: 10.1109/PAAP.2018.00018.

[44] R. Bellman, R.E. Bellman, and Rand Corporation. *Dynamic Programming*. Rand Corporation research study. Princeton University Press, 1957.

[45] Nvidia. *Volta Architecture White Paper*. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf (visited on 07/04/2019).

[46] B. Le Gal and C. Jego. “High-Throughput LDPC Decoder on Low-Power Embedded Processors”. In: *IEEE Communications Letters* 19.11 (Nov. 2015), pp. 1861–1864. ISSN: 1089-7798. DOI: 10.1109/LCOMM.2015.2477081.