TOR VERGATA
UNIVERSITY OF ROME

Doctoral Dissertation

Doctoral Program in Computer Science, Control and GeoInformation

Towards resilient and effective Network Services

Marco Faltelli

Supervisors
Prof. Francesco Quaglia
Prof. Giuseppe Bianchi

Doctoral Program Coordinator
Prof. Francesco Quaglia

Academic year 2022/23
(XXXV PhD cycle)

Abstract

In recent years, high-performance polling frameworks like DPDK have been proposed to enable fast packet processing. Despite the remarkable improvements, such frameworks still provide some questions to be addressed. A first well-known drawback is the cost of precious CPU resources dedicated to continuously poll the NICs, coupled with the lack of resilience in resource-sharing contexts like public clouds. Moreover, as NICs are about to reach the Terabit Ethernet, questions about the multi-queue scaling must be asked, as well as whether a better architecture can be designed in order to deliver predictable latency to applications.

In this scenario, this Dissertation aims to tackle these limitations by exploiting techniques like multi-core low-overhead coordination, parallel computation, queuing theory, and dynamic scaling. More specifically, two contributions are presented.

Firstly, this Dissertation proposes an approach, descriptively called Metronome, which has the dual goals of providing CPU utilization proportional to the load and allowing flexible sharing of CPU resources between I/O tasks and applications. Metronome replaces the continuous polling with an intermittent sleep&wake mode and revolves around a new multi-threaded operation, which improves service continuity. Since the proposed operation trades CPU usage with buffering delay, we propose an analytical model devised to adapt the sleep&wake parameters to the actual traffic load dynamically, meanwhile providing an average target latency. Our experimental results show a significant reduction in CPU cycles, improvements in power usage, and robustness to CPU sharing even when challenged with CPU-intensive applications.

Secondly, an algorithm for a non-blocking, parallel driver is presented. By exploiting read-modify-write instructions, the algorithm enables a new scale-up policy for net-

work stacks; the novelty behind this is not only in the software coordination among the threads but also in a transparency mechanism that allows the algorithm to run with unmodified NICs. This policy permits new possibilities for scaling in hundred-gigabit NICs and has the inherent benefit of improving both mean and tail latency.

Contents

"ci si fa grandi resistendo ad una sventura ed espiando le proprie colpe,
e si diventa invece piccoli gonfiandosi con le menzogne
e facendo risorgere i cattivi istinti per il fatto di vincere"

Giuseppe Prezzolini

Chapter 1

Introduction

1.1 Context

1.1.1 Data Centers

Data Centers are one of the most challenging contexts in modern Computer Engineering: many users execute their tasks on the same servers, and each of them has the illusion of being constantly run. In practice, it is known that these users share a various number of resources: the CPUs for executing tasks, as well as caches and RAM for storing relevant data, the PCIe transactions to send and get data from the NIC, and the available bandwidth in the network.

Sharing resources in cloud contexts can cause interference among applications running on the same physical or virtual machines. The reason behind this is that the cloud infrastructure often employs multi-tenancy, where multiple applications from different users are hosted on the same set of resources. When resources are shared, there is a possibility that one application may consume more resources than it needs, causing other applications to suffer from resource starvation. Similarly, an application that consumes excessive CPU cycles may cause other applications to experience delays or timeouts; this is especially harmful in the case of latency-sensitive applications.

Furthermore, applications may use inefficiently these resources for a number of rea-

sons: at the code level, poor optimization, lack of parallelism, or outdated software could be a bottleneck. Inappropriate resource allocation, lack of workload balancing, or bursty workloads can limit performance too.

From the provider's perspective, it becomes critical to make a careful, flexible, and smart usage of these resources. In fact, the explicit goal of multi-tenancy is to pack more and more applications into the same server to reach peak capacity and maximize gains[1]. Indeed, Google states that even minor improvements in resource utilization can save millions of dollars [2].

Takeaway 1: Cloud providers strive to make the maximum out of their hardware, so intelligent and careful use of shared resources (CPU, RAM, network bandwidth) must be set up.

1.1.2 The need for CPU cycles

For decades, application improvement has intensely relied on two different laws regarding computer architecture and chip performance: Moore's Law and Dennard Scaling.

- **Moore's Law** is a prediction made by Intel co-founder Gordon Moore in 1965 that the number of transistors on a microchip would double approximately every 18 to 24 months, leading to a corresponding increase in the processing power of computers.

- **Dennard Scaling** is a term used to describe the phenomenon of transistors becoming smaller and more densely packed on a microchip while consuming less power simultaneously. Consequently, the power consumption per mm^2 would remain roughly constant. Robert Dennard first observed this phenomenon in the 1970s.

These two predictions have held true for several decades and have been a driving force behind the rapid advancement of technology. However, the end of Moore's Law and Dennard Scaling seems to be approaching [3].

One of the main reasons for this is the physical limitations of transistors. As transistors continue to shrink in size, they are approaching the limit of what is possible with current technology. This is due to the fact that as the transistors get smaller, their electrical properties begin to change, resulting in increased leakage current and higher power consumption. Additionally, as the transistors get smaller, they become more susceptible to thermal effects, leading to increased power consumption.

Another factor is the increasing cost of designing and producing smaller transistors. As the size of transistors decreases, it becomes increasingly expensive to design and manufacture them.CPUs are also known for being the most greedy component in terms of power consumption [4]: Barroso and Hölzle [5] show that in a Google datacenter, CPUs account for 33% of the peak power usage of a server.

Accelerators like GPUs, TPUs, or programmable hardware like FPGAs have been proposed to offload computation from the CPUs and save cycles. However, there is no one-size-fits-all accelerator, as some workloads perform better on CPUs or either cannot be offloaded because of the lack of primitives. Moreover, these devices are still conceived as peripherals, while the CPU keeps its role as the primary, central device for computation.

Takeaway 2: CPU cycles are precious and we should make an effort to efficiently use these cycles, as while CPU improvements stall, applications ask for more computing power than ever. At the same time, it is also crucial to implement power-saving policies to make data centers more sustainable.

1.1.3 The network perspective

Data center networks are the backbone of modern computing, connecting servers, storage devices, and other networked equipment in order to provide fast and reliable communication between devices. These networks play a critical role in the functioning of data centers, which organizations of all sizes use to host their applications and data.

Rate trends

One of the critical challenges in designing a data center network is dealing with a large amount of data that needs to be moved. With the increasing amount of data being generated and consumed by modern applications, data center networks need to be able to handle high-bandwidth traffic while maintaining low latency. This is typically achieved by using high-speed networking technologies: in the last years, new technologies have emerged, like Infiniband and RDMA, and there has been a massive improvement in network speeds. In fact, Network Interface Cards (NICs) have now reached 400Gbps [6], and higher speeds like 800Gbps, and 1.6Tbps (the so-called Terabit Ethernet) are expected to become an IEEE standard between 2023 and 2025 [7]. Considering that in 2012 the standard NICs could support 10-40Gbps rates [8], this implies NICs have had a 10-40x improvement in the last 10 years, and the consequence is increased pressure on CPUs, as they have to process more and more packets in the same period of time.

Software trends

Because of the motivations explained in Sections 1.1.2 and 1.1.3, it is clear that computer scientists currently find themselves in the following situation: while NICs are rapidly increasing their throughput, the CPUs in charge of processing this traffic are

at a stagnation point and struggle to process increasing traffic rates. One of the main solutions was to re-architect the way packets are processed in software. In fact, traditional in-kernel processing is anything but tailored for modern requirements because of the complex network stack that packets have to cross, as well as the system calls and context switches.

For these reasons, Kernel bypass has emerged: the name refers to a technique used in computer systems to bypass the kernel, or the central part of an operating system that manages system resources and controls interactions between hardware and software. Bypassing the kernel can be done for a variety of reasons, such as to improve performance, reduce latency, or to access low-level hardware features that are not directly exposed by the kernel. There are a few different methods to achieve kernel bypass, including special kernel modules, userspace networking libraries, or direct access to network interfaces. The main framework currently used is the Data Plane Development Kit (DPDK) [9].

Application requirements

Data Center applications rely heavily on distributed architectures and microservices, as different components interact with each other exchanging data through the network. A common way to properly handle a user's request is to split it into different sub-requests, distribute them onto a set of leaf nodes, and subsequently merge the responses. For the application to be responsive, it is crucial that these sub-requests are completed in a short period of time. As requests are heavily parallelized, the response time is bounded by the slowest of the sub-request's response time; therefore, it is paramount to build systems that can achieve not only a very low latency but, most of all, a predictable one [10]. Latency predictability in data centers has been

widely studied [10, 11], and there are many possible sources of tail latency, like resource sharing, queuing, and power management. In more detail, the literature shows that we have two sides of the same coin. On one side, we have to deal with latency variability while dealing with complex systems such as modern calculators and cannot get entirely rid of it. On the other one, it is crucial to design architectures explicitly built to mitigate it as much as possible.

Takeaway 3: There is a growing need for high-performing, kernel-bypass frameworks which must provide (1) high throughput and (2) low, predictable latency.

1.2 Problem statement

It is clear to the reader that this is quite a tangled context, where different software and hardware trends have brought different approaches and technologies that need to be melted together. This causes different problems, which are the focus of this dissertation:

1. **100% CPU consumption** of the cores assigned to kernel-bypass frameworks like DPDK, as they need to poll the NICs to check for incoming packets constantly. This counteracts Takeaways 1 and 2 and is exacerbated by the fact that while data center networks are designed to handle peak load, they are largely underutilized: Microsoft reveals that 46-99% of their rack pairs exchange no traffic at all [12]; at Facebook, the utilization of the 5% busiest links ranges from 23% to 46% [13], and [14] shows that the percentage of utilization of core network links (by far the most stressed ones) never exceeds 25%.

2. **Lack of resilience in cloud contexts.** Major cloud providers have enabled

kernel-bypass software in their datacenters [15, 16]. Still, such frameworks are designed to run in an isolated fashion, with exclusive access to any resource, and therefore they do not seem suited for CPU-sharing contexts like public clouds.

3. **Lack of scalability:** as Golestani, Mirhosseini et al. showed in [17], current frameworks lack scalability in terms of many queues and cores; in fact, poor scaling is caused by the increased use of memory, which leads to increased cache pressure and therefore, degradation in peak throughput and higher average latency. This limitation will be more and more significant as we move towards the Terabit Ethernet and beyond (see Section 1.1.3 and Takeaway 3), where a highly efficient scaling to many cores is needed.

4. **Lack of frameworks specifically designed to deliver low and predictable latency:** it is well known that scale-up architectures are the best for delivering lower and tail-bounded latency (see Section 6.1). Still, such frameworks exploit scale-out mechanisms to avoid synchronization overheads in sharing a single receive queue [17]. This causes missing Takeaway 3.

1.3 Objective

The goal of this dissertation is the design and implementation of novel policies and mechanisms in the context of high-performance networking, which can provide an answer to the above-mentioned problems. More specifically, the author has focused on the following approaches:

1. **Adaptive polling mechanism** to replace the static, 100% CPU consumption approach with an adaptive one, where threads can go to sleep when there is no traffic to be processed. An adaptive model fine-tunes this sleep period, choosing

how much each thread needs to sleep based on the incoming traffic. This permits freeing CPU cycles to other applications while avoiding too big sleep periods and, therefore, packet loss. This solves Problem 1 and, as a side effect, has a positive impact on reducing energy consumption.

2. **Multi-threaded architecture** with low-level coordination among threads to enable enhanced robustness in CPU-sharing contexts. Different threads can share a receive queue and race for exclusive access, and this approach brings increased resilience, thus solving Problem 2.

3. **Scale-up network stacks** instead of the current scale-out policies. More threads can process the same receive queue in parallel without any delay or waiting period. The reason behind moving to a scale-up network stack is that it brings benefits, especially in terms of low and tail-bounded latency. A first consequence of this approach is that one queue is enough to process packets coming at very high gigabit rates, thus reducing the number of queues needed and overcoming Problem 3. To the best of the author's knowledge, this is the first networking framework explicitly designed at the architecture level to provide low and tail-bounded latency, solving Problem 4.

1.3.1 Scope

This dissertation focuses explicitly on the DPDK framework for a number of reasons outlined here:

- Ease of writing code in a user-space context and testing it straight away. The opposite solution of writing in-kernel code would have required many limitations, like a limited range of action and complex deploying strategies and debugging.

- DPDK delivers the best performances compared to other solutions [18].
- it is a well-documented framework and constantly maintained.
- it is widely used by both industry and academia.

The author wants to underline that many of the contributions proposed in this dissertation are not restricted to the DPDK community. In fact, concepts like concurrency, adaptive models, and queuing theory can be put in place also in other solutions, like the Linux kernel or RDMA. Implementing similar solutions in such contexts would likely be more complex: as stated before, the main challenge for the Linux kernel would be testing and deploying the code in kernel space, as this may cause stability issues or system crashes; also applicability to an interrupt-based solution should be kept in consideration. For RDMA, the limited documentation and restricted knowledge from the community is the main limitation at the moment. It is also necessary to underline that Emmerich et al. paved the way for part of this dissertation by comprehensively explaining how DPDK-style drivers work [19]. The lack of similar work applied to the Linux kernel or RDMA drivers is definitely a limitation at the moment. The author hopes that this dissertation might be an inspirational starting point for some readers toward this goal.

1.4 Dissertation outline

This dissertation is organized with the following outline:

- Part I presents Metronome, an approach devised to replace the static, fixed 100% CPU consumption of polling frameworks with an adaptive sleep&wake approach, as well as increased robustness. Metronome meets Objectives 1 and 2 in Section 1.3.

- Part II presents a scale-up mechanism for network stacks, as opposed to the current scale-out policy. This approach permits vertical scaling by binding more CPU cores to the same receive queue, positively impacting mean and tail latency because of its inherent work-conserving characteristic. This work meets Objective 3 in Section 1.3.

1.5 Published Works

During my PhD, I've been the first author and co-author of the following publications and technical reports:

- Marco Faltelli, Giacomo Belocchi, Francesco Quaglia, Salvatore Pontarelli, and Giuseppe Bianchi. "Metronome: Adaptive and Precise Intermittent Packet Retrieval in DPDK". in: *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '20. Barcelona, Spain: ACM, 2020, pp. 406–420. DOI: 10.1145/3386367.3432730

- Marco Faltelli, Giacomo Belocchi, Francesco Quaglia, Salvatore Pontarelli, and Giuseppe Bianchi. "Metronome: Adaptive and Precise Intermittent Packet Retrieval in DPDK". in: *IEEE/ACM Transactions on Networking* (2022), pp. 1–15. DOI: 10.1109/TNET.2022.3208799

- Marco Faltelli, Giacomo Belocchi, Giuseppe Bianchi, and Francesco Quaglia. *A non-blocking, parallel driver for low and predictable latency*. Tech. rep. Consorzio Nazionale Interuniversitario per le Telecomunicazioni - CNIT, 2022

- Valerio Bruschi, Marco Faltelli, Angelo Tulumello, Salvatore Pontarelli, Francesco Quaglia, and Giuseppe Bianchi. "Offloading Online MapReduce tasks with

Stateful Programmable Data Planes". In: *IEEE NETPROC 2020*, pp. 17–22. DOI: 10.1109/ICIN48450.2020.9059417

- Giuseppe Bianchi, Marco Faltelli, and Valerio Bruschi. "Back to the Future: Towards Hardware "Netputing" Architectures". In: *IEEE MedComNet 2020*, pp. 1–4. DOI: 10.1109/MedComNet49392.2020.9191475

Furthermore, the "*A non-blocking, parallel driver for low and predictable latency*" paper is about to be submitted for revision to an ACM conference.

1.6 Conferences Attended

In these three years, I've attended the following conferences:

- ACM CoNEXT 2020 (remote), where I presented a first version of Metronome.
- IEEE ICIN 2020 (Paris, France), where I presented the MapReduce paper at the NETPROC workshop.
- USENIX OSDI 2021 (remote) as an attendee.
- ACM HotNets 2021 (remote) as an attendee.
- ACM CoNEXT 2022 (Rome, Italy) as an attendee.

1.7 Rewards and Internships

This dissertation is supported by a Microsoft PhD Fellowship award. As part of the fellowship, I had the opportunity to spend three months at Microsoft Research Cambridge, UK, during Summer of 2022 for an internship. I've been part of project Silica, targeting path optimization in the context of Cloud Infrastructure robotics. The work was done under the supervision of Dr. Sergey Legtchenko.

Chapter 2

Background

This chapter focuses on the most relevant concepts and technologies to this dissertation. The author first presents the state of the art and then, in the last section of the chapter, the main difference between these and the novel technologies presented in this dissertation. This chapter partially includes figures and verbatim copies of the text from the author's papers.

2.1 NIC-CPU interaction

When dealing with how the NIC and the CPU communicate for handling packets arrival, there are two possible ways of interaction:

- **Interrupt-based** solutions are based on the following schema: every time a packet arrives, the NIC informs a CPU core of this event by sending an interrupt, and consequentially the CPU core will retrieve the incoming packet. One of the advantages of this approach is that CPU cores can be utilized for other tasks when there is no incoming traffic, as the NIC is in charge of alerting them. This is the approach used in modern operative systems.

- **Polling-based** solutions use the opposite approach: the CPU constantly checks the shared memory area with the NIC for incoming packets without any warning

by external components. This approach has the great advantage of not paying the fixed price of one interrupt per packet and therefore achieves the best latency, but costs in terms of sacrificing one or more cores to this constant polling operation. This approach is used in high-performance specific frameworks like DPDK [9].

The recent advantages in NIC speeds and the stagnation of CPU performances because of the end of Moore's law and Dennard Scaling have brought to evidence the limitations of an interrupt-based approach. In fact, interrupt-based solutions suffer from the latency brought by the system calls used to interact with the kernel-level driver managing interrupts, packet copies to user space, and so on. Moreover, an interrupt-based architecture operating at extreme interrupt arrival speed may cause livelocks [25]. The Linux NAPI subsystem [26] aims at tackling these limitations by providing a hybrid approach which tends to eliminate receive livelocks by dynamically switching between polling and interrupt-based packet processing, depending on the current traffic load; the polling period has a maximum budget, either in terms of time or number of packets processed.

For the above-mentioned reasons, polling-based solutions have been widely adopted in the last few years, with DPDK being the most known both in industry and academia.

2.1.1 Network Drivers

It is now essential to focus on network drivers in order to understand how the data movement between NIC and CPU actually happens. A network driver is no more than a piece of software built to move packets from the NIC memory to the main memory and vice versa. The NIC and the CPU share one or more circular buffers also called Receive (RX) or Transmit (TX) queues; incoming traffic can be split into multiple RX

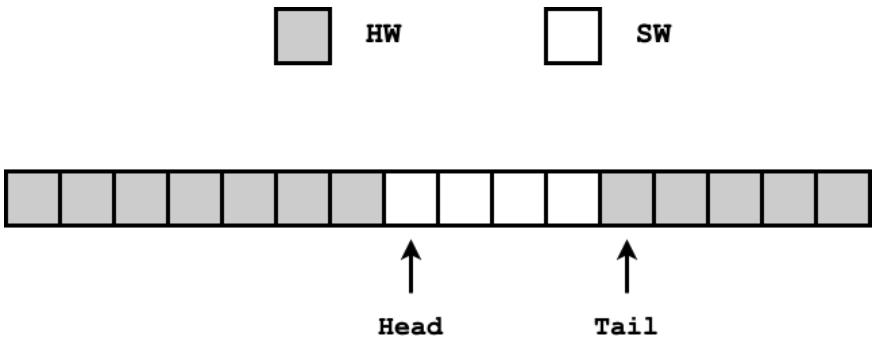


Figure 2.1: A typical ring buffer

queues through filters or a hashing algorithm, while multiple TX queues are usually merged on the NIC. These queues are composed of descriptors, each containing some metadata about the packet and a pointer to a memory area where the packet is located. An RX queue routine is roughly like this (the TX one is specular): the NIC controls the area between the tail and the head of the queue [27] (see the grey boxes in Figure 2.1) and moves the head forward for every descriptor it fills. Complementarily, the software controls the rest of the queue, swapping the NIC-populated descriptors with empty ones and moving the tail. The software can understand whether a descriptor is populated or not through the DD bit, which the NIC sets once it fills the descriptor.

Rx side flow

For the reader's convenience, it is now shown a simplified routine of the receive side of a network driver in Listing 2.1.1. First, the driver retrieves the relevant pieces of information, namely the shared buffer with the NIC (line 6) and the descriptor from where it left off at the last iteration (line 7). The driver checks how many of the following descriptors were populated by hardware by reading their DD bit (line 10) up to a fixed batch value (usually 32). Each of these descriptors is moved to a user-space

Listing 2.1: A standard receive function of a network driver

```
1  #define wrap_ring(index) (uint16_t) (index % RING_SIZE)
2
3  uint32_t ixgbe_rx_batch(struct device* dev, uint16_t queue_id, struct
       pkt_buf* bufs[]) {
4    //Get the queue struct for device dev and queue queue_id
5    struct ixgbe_rx_queue* queue = get_queue(dev, queue_id);
6    struct pkt_buf* buffer = queue->buffer;
7    uint16_t rx_index = queue->rx_index; // descriptor index we checked in
         the last run of this function
8    uint16_t last_rx_index, i;
9    for (i = 0; i < BATCH_SIZE; i++) {
10     if (!(buffer[rx_index] && DD_BIT))
11     //Descriptor has not been filled by the NIC yet, exit the loop
12       break;
13     else {
14       //Copy the descriptor in the user space buffer
15       bufs[i] = buffer[rx_index];
16       //Replace the descriptor with a new one from the mempool
17       buffer[rx_index] = mempool_desc_get();
18       last_rx_index = rx_index;
19       rx_index = wrap_ring(rx_index + 1);
20     }
21   }
22   //Free the processed descriptors back to the NIC
23   set_register(TAIL, dev, queue_id, last_rx_index)
24 }
```

buffer (line 15) and replaced with a new one taken from a pre-allocated memory pool (line 17). The software can now update the tail, moving it by the number of packets it has retrieved (line 23).

2.2 Network stacks and frameworks

The network stack is the portion of code in charge of implementing the different network layers, as well as the different functionalities. Two different approaches can be pursued:

- **kernel-based** network stacks are solutions where all traffic passes through the kernel, which is in charge of implementing all the network functionalities and exposing a set of primitives to the users in order to send/receive data. OS kernel

stacks are the standard solution for PCs and servers, as they already offer all the functionalities users require. While these stacks are mostly interrupt-based, some recent polling frameworks (e.g. netmap [28], PFQ [29] and PF_RING ZC [30]) rely on kernel drivers. XDP [31] has been recently proposed as a part of the eBPF [32] project to process packets as soon as they get to the software before any kernel allocation or routine is invoked.

- **kernel-bypass** stacks, like DPDK [9], tend to completely bypass the kernel, thus interfacing directly with the underlying NIC(s). This solution has the advantage of removing all the kernel latencies and inefficiencies typical of a widely generic solution to specialize for performances like throughput and latency.

2.2.1 Queue modeling

It is interesting to have a look at the different queuing models and how they are exploited in current network stacks. We typically use Kendall's notation, a widely used system for representing queuing models, which provides a concise and standardized way to describe the key characteristics of a queuing system. Kendall's notation consists of three parts:

$$A/B/c \tag{2.2.1}$$

- **Arrival Process (A)**: The arrival process describes how customers arrive at the system. It is represented by a single letter that indicates the distribution of inter-arrival times between consecutive customers.

- **Service Time Distribution (B)**: The service time distribution represents the time required to serve a single customer. Like the arrival process, it is denoted by a single letter indicating the distribution of service times.

- **Number of Servers (c)**: The number of servers in the system is represented by a numerical value. It indicates how many customers can be served simultaneously. If this value is omitted, it is assumed to be 1 (single server). Examples of server numbers are: 1 (single server), 2 (two servers), ∞ (infinite number of servers, no waiting).

Typical Arrival Process (A) and Service Time (B) distributions are the following:

- **M**: Markovian (or Poisson) process (exponential inter-arrival times);
- **D**: Deterministic arrivals (fixed inter-arrival times);
- **G**: General distribution for inter-arrival times.

We will focus here on the case where the arrival process is Markovian (M) and the service time general (G).

Modern network stacks are organized through a variation of the $M/G/1$ model, namely the $k \times M/G/1$ model. By considering the Rx driver queues as the model queues, we have k queues operating independently with one servant (thread) each. The incoming traffic rate λ is (theoretically) split equally among the queues, and each thread processes the incoming traffic at rate μ. In this case, performances are improved by adding more independent Rx queues at the driver level, as well as more threads. This scaling option is the so-called **scale-out** method (see left side of Figure 2.2) and it is the model currently used in every modern network stack. A reason for using this model is that modern NICs permit redirection of the packets belonging to the same flow onto the same queue, therefore avoiding reordering possibilities.

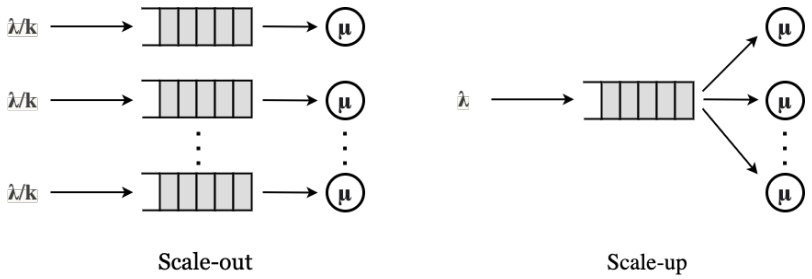


Figure 2.2: Scale-out vs. Scale-up policy

2.2.2 DPDK

This dissertation focuses on the DPDK (Data Plane Development Kit) framework for the reasons outlined in Section 1.3.1. It is a kernel-bypass framework explicitly designed to optimize network functions by enabling high-performance packet I/O and efficient data plane processing in multi-gigabit scenarios. While NFV aims to virtualize network functions on general-purpose servers, this virtualization introduces overhead that can impact performance. DPDK helps mitigate this by providing a highly optimized user-space packet processing framework: it includes optimized libraries and drivers that enable fast packet I/O, advanced packet classification, and efficient data manipulation for typical network functions such as routers, firewalls, load balancers and so on. DPDK is managed by the Linux Foundation and constantly maintained by the most relevant companies in the field of NFV.

2.3 Non-blocking algorithms

Non-blocking algorithms are a type of concurrent programming technique used to design systems that can handle multiple tasks or processes simultaneously without the possibility of blocking each other.

In traditional blocking algorithms, a process or task may block or wait for another process to complete before proceeding. This typically happens when exclusive access

to specific resources is granted (like in the case of locks, mutex, and semaphores), leading to inefficiencies and delays in the system, especially when multiple processes are involved.

On the contrary, non-blocking algorithms allow multiple processes to proceed independently without waiting for each other. Each process operates on its own copy of the data, while the algorithm ensures that conflicts or collisions are resolved without blocking or waiting for other processes.

Non-blocking algorithms can improve the performance, scalability, as well as responsiveness of concurrent systems, particularly in high-load and distributed environments. They are mainly based on specific hardware support from CPUs for read-modify-write (RMW) operations like compare-and-swap (CAS), load-link/store-conditional, or transactional memory [33, 34].

2.4 Where this dissertation fits

2.4.1 Metronome

As of Section 2.2, Metronome is developed in a polling environment like DPDK but presents CPU-proportional features similar to XDP. The evaluation in Chapter 5 widely compares Metronome, static DPDK polling and XDP.

Metronome's scale-out policy is slightly different than the standard one: in fact, the queue-to-thread binding is not fixed but rather dynamic, as it depends on which thread wins the race for a certain queue (See Figure 2.3).

2.4.2 Non-blocking, single-queue driver

The main difference with respect to the state of the art is the deployment of a **scale-up** policy instead of the current scale-out mechanism presented in Section 2.2.1 (see

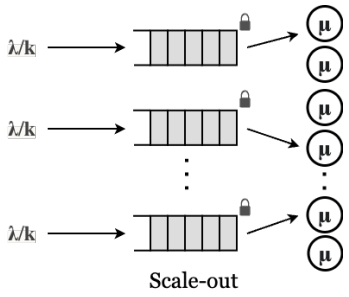


Figure 2.3: Metronome's variation of the scale-out policy

Figure 2.2 on the right). The reason behind this is that a single queue shared among different threads enables global visibility of the workload to be processed by all the threads, therefore implying a work-conserving policy for network stacks and, consequently, minimizing mean and tail latency (see Section 6.1 for a broader explanation with practical results from simulations).

The design and implementation of a shared queue among threads are made possible through non-blocking instructions and principles like the one presented in Section 2.3. The introduction of such instructions and their HW support is quite recent, and this is one of the reasons why this approach was not explored before: without these instructions, the only possibility of a scale-up system would have been through a lock mechanism, which of course would have caused significant performance limitations. As said, the networking community hardly knows the approaches presented in 2.3, and the author hopes that this dissertation can be a first approach in bridging the gap between the two worlds.

Part I

Metronome

Introduction

In this Part, the author presents Metronome, a novel solution aimed at concurrently achieving two objectives: providing CPU utilization proportional to the workload and enabling the flexible sharing of CPU resources between I/O tasks and applications. These two joint goals are set to overcome Problems 1 and 2 presented in Section 1.2. Metronome is an approach devised to replace the continuous DPDK polling with a sleep&wake intermittent approach. Albeit this might seem in principle an obvious idea, its advantages are linked to several factors that the author copes within this part. First, a proper implementation/usage of sleep&wait operating system services needs to be put in place. As for this aspect, Metronome can work effectively by relying on microsecond-level sleep phases supported by either the Linux `nanosleep()` service or an own new service called `hr_sleep()`. The latter offers a few advantages and is also independent of limitations related to system parameterization and thread priorities. Second, Metronome revolves around a novel architecture and operating mode for DPDK, where incoming traffic, from either a single receive queue or multiple ones, is shared between multiple threads through a `trylock`—as it will be discussed this also offers advantages by the side of robustness versus operating system thread-scheduling decisions. These threads dynamically switch—in a coordinated manner—from polling the receive queue to sleep phases for short and tunable periods of time when the queue is idle. Owing to a suitable adaptation strategy that tunes the sleeping times

depending on the load conditions, Metronome achieves a stable tunable latency and no substantial packet loss difference compared to standard DPDK while reaching a significant reduction for both CPU usage and power consumption. This part is divided into three chapters:

1. Chapter 3 presents Metronome's architecture, in particular with respect to the engineering of a high-performance sleep service (and its comparison to Linux' `nanosleep()`) and the multi-threaded model to enhance robustness.

2. Chapter 4 describes Metronome's adaptive model, which has the goal of fine-tuning the sleep period depending on the actual traffic load, which is estimated at runtime.

3. Chapter 5 presents a deep evaluation of Metronome, focusing on different performance indicators like CPU consumption, power consumption, latency, throughput, and so on, as well as the comparison with similar solutions like static DPDK polling and XDP.

A first version of Metronome has been presented at *ACM CoNEXT 2020*. An extended version was also published on *IEEE/ACM Transactions on Networking* in October 2022. Metronome is also publicly available at [35]. This part partially includes figures and verbatim copies of the text from the papers presenting Metronome [20, 21].

Chapter 3

Metronome architecture

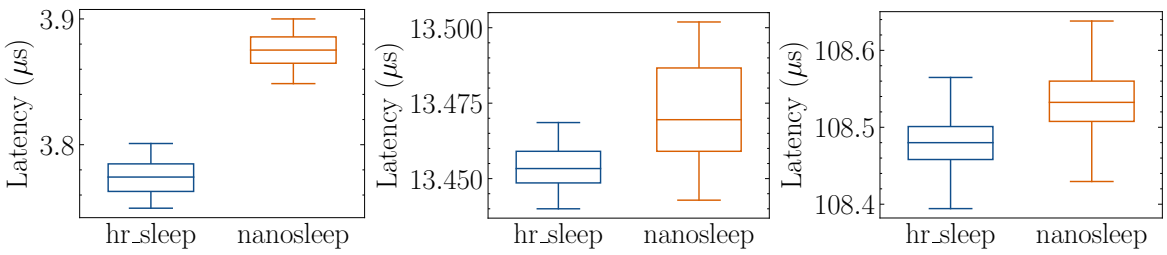
3.1 Fine-grained Thread Sleep Service

The precision of the thread-sleep interval supported by the operating system, is essential for the construction of any solution where the following two objectives need to be jointly pursued: 1) threads must leave the CPU if there is currently nothing to do (in our case by the side of packet processing); 2) threads must be allowed to be CPU rescheduled—gaining again control of the CPU—according to a tightly controlled timeline. Point 2) would allow the definition of an architectural support where we can be confident that threads will be able to be CPU dispatched exactly at (or very close to) the point in time where we would like to re-execute a poll operation on the state of a NIC—to determine whether incoming packets need to be processed. On the other hand, point 1) represents the basis for the construction of a DPDK architecture not based on full pre-reserving of CPUs to process incoming packets.

In current conventional implementations of the Linux kernel, the support for (fine-grained) sleep periods of threads is based on the `nanosleep()` system call. A few limits of this service are related to its dependency on a slack factor assigned to threads, which is checked when they request to sleep. This factor can be controlled using the `prctl()` system call, putting it to the minimal value of 1. If such a setting is not adopted,

then for any thread that is not in the real-time CPU-scheduling class we have at least 50 microseconds as the slack imposed by the Linux kernel, which makes the awake of the thread less controllable in terms of precision under fine-grained sleep requests. Furthermore, when entering the kernel level execution of `nanosleep()`, the Thread Control Block (TCB) is checked because of the need to determine the current slack value, which makes the service run a few machine instructions to reconcile the real value to be adopted in the sleep phase with the information kept in the TCB.

While developing Metronome, we also implemented an alternative sleep service, namely `hr_sleep()`, mountable through an external kernel module. This variant fully avoids any interaction with thread management information at the kernel level (such as the current slack value kept in the TCB). Hence, it also avoids running the additional machine instructions needed to manage this information, for any CPU-scheduling class of threads (real-time or not).In any case, in Figure 3.1 we show that the performance of `hr_sleep()` is completely similar to the one of `nanosleep()` under the scenario where the latter is configured with the minimal admissible slack currently supported by the Linux kernel. As said, this setting of kernel-level parameters is not requested at all when using `hr_sleep()`. The tests have been conducted on an isolated NUMA node equipped with Intel Xeon Silver 2.1 GHz cores. The server is running Linux kernel 5.4. We have run an experiment where a million samples of the wall-clock-time elapsed between the invocation of the sleep service and the resume from the sleep phase are collected. This wall-clock-time interval has been measured via start/end timer reads operated through `__rdtscp()`. We show the boxplots for both the sleep services with different timer granularity requests (1, 10, 100 μs). These data have been collected by running the thread issuing the sleep request as a classical `SCHED_OTHER` (normal) priority thread and—as hinted before—with the timer slack of



(a) 1 μs (b) 10 μs (c) 100 μs

Figure 3.1: Boxplots for `hr_sleep()` and `nanosleep()` latencies

`nanosleep()` set to 1μs. The results show that `hr_sleep()` provides some minimal gains both in terms of mean latency and variance even under such extreme setting of the slack value for `nanosleep()`. In any case, the avoidance of the reliance on kernel-level parameters makes `hr_sleep()` fully independent of any kernel configuration choice for the minimal admissible slack.

3.2 Actual Thread Operations

In this section, we describe how threads in charge of processing packets operate in Metronome. To this end, let us start with a brief discussion of the state-of-the-art DPDK architecture: on the receiving side, NICs may convey their incoming traffic into a single Rx queue or split such traffic into multiple Rx queues through Receive Side Scaling (RSS). A DPDK thread normally owns (and manages) one or more Rx queues, while an Rx queue belongs to (namely, is managed by) one DPDK thread [36]. Therefore, the behavior of a DPDK thread is no more than an infinite `while(1)` loop in which the thread constantly polls all the Rx queues it is in charge of. This approach raises some important shortcomings such as (i) greedy usage of CPU even in light load scenarios and (ii) prevention of any Rx queue sharing among multiple threads. As for point (ii) we note that in 40GbE+ NICs, queues experience heavy

loads despite the use of the RSS feature, e.g. on a 100Gb port with 10 queues, each queue can experience 10Gb rate traffic or even more. Preventing multi-threaded operations on each single Rx queue, and the exploitation of hardware parallelism for processing incoming packets from that queue, looks therefore to be another relevant limitation (see Section 5.6).

Compared to the above described classical thread operations in state-of-the-art DPDK settings, we believe smarter operations can be put in place by sharing a Rx queue among different threads and putting these threads to sleep, when queues are idle, for a tunable period of time, depending on the current traffic characteristics. In other words, via a precise fine-grained sleep service, and lightweight coordination schemes among threads, we can still control and improve the trade-off between resource usage and efficiency of packet processing operations.

To this end, the `hr_sleep()` service has been coupled in Metronome with a multi-threaded approach to handle the Rx queues. In more detail, in our DPDK architecture we have multiple threads that sleep (for fine grained periods) and then, upon execution resumption, race with each other to determine a single winner that will actually take care of polling the state of some Rx queue for processing its incoming packets. In this approach we do not rely on any additional operating system services to implement the race; rather, we implemented the race resolution protocol purely at user space via atomic Read-Modify-Write instructions, in particular the `CMPXCHG` instruction on `x86` processors, which has been exploited to build a lightweight `trylock()` service. The race winner is the thread that atomically reads and modifies a given memory location (used as the lock associated with an Rx queue), while the others simply iterate on calling our new `hr_sleep()` service, thus immediately (and efficiently, given the reduced CPU-cycles usage of `hr_sleep()`) leaving the CPU—given that another

thread is already taking care of checking with the state of the Rx queue, possibly processing incoming packets. Interested readers can have a look at Section 3.3 for a basic coding example of DPDK-traditional and Metronome approaches.

We also note that using multiple threads according to this scheme allows creating less correlated awake events and CPU-reschedules, leading to (i) more predictability in terms of the maximum delay we may experience before some Rx is checked again for incoming packets and (ii) less work to be done for each thread, since the same workload is split across more cores. This is true especially when the CPU-cores on top of which Metronome threads run are shared with other workload. In fact, the multi-thread approach reduces the per-CPU load of Metronome. This phenomenon of *resiliency* to the interference by other workloads will be assessed quantitatively in Section 5.6, along with the benefits for the applications sharing the same cores with Metronome.

Overall, with Metronome we propose an architecture where Rx queues can be efficiently shared among multiple threads (see Figure 3.2): to each queue corresponds a lock which grants access to that queue. Threads can acquire access to a queue through our custom `trylock()` implementation, which provides non-blocking and minimal latency synchronization among them. For each of its queues, every thread tries to acquire the corresponding lock, and passes to a different queue chosen randomly (see Section 4.5) if lock acquisition fails. Otherwise, if the thread wins the lock race it processes that queue as long as there are still incoming packets, then it releases the lock once the queue is idle. Once a thread has processed (or at least has tried to process) the Rx queues, it can go to sleep for a period of time proportional to (and controllable in a precise fine-grained manner depending on) the traffic weight it has experienced during its processing. Scheduling an awake-timeout through a fine-

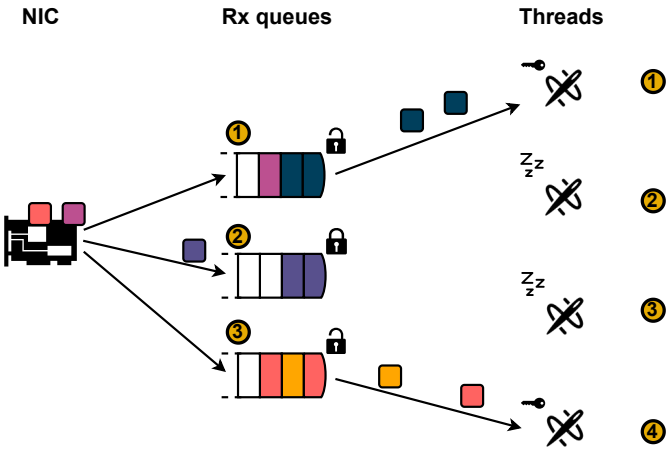


Figure 3.2: Metronome architecture

Listing 3.1: A standard DPDK polling loop

```
1 curr_queue = THREAD_ASSIGNED_QUEUE;
2
3 while (1) {
4   nb_rx = receive_burst(queue[curr_queue], pkts, BURST_SIZE);
5   if (nb_rx == 0)
6     continue;
7   process_and_send_pkts(pkts, nb_rx);
8 }
```

grained sleep service enables very precise and cheap thread-sleep periods, which are essential at 10Gb+ rates, and can still provide resource savings at lower rates. How a thread can elicit an awake-timeout period without incurring an Rx queue filling is carefully explained through our analytical model in Chapter 4. This model is used to make the Metronome architecture self-tune its operations, providing suited trade-offs between resource usage (CPU cycles and energy) and packet processing performance.

3.3 Skeleton code

We briefly compare the classical and Metronome methods through a simplified (yet meaningful) code example of a typical DPDK thread routine. This example only

Listing 3.2: Our novel DPDK processing loop

```
1  curr_queue = THREAD_ASSIGNED_QUEUE;
2
3  while (1) {
4    if(!trylock(lock[curr_queue])) {
5      curr_queue = randint(n_queues);
6      hr_sleep(timeout_long);
7      continue;
8    }
9
10   while(nb_rx = receive_burst(queue[curr_queue], pkts, BURST_SIZE))
11     process_and_send_pkts(pkts, nb_rx);
12   unlock(lock[i]);
13
14   hr_sleep(timeout_short);
15 }
```

focuses on the different coding approaches, rather than other aspects (e.g., implementing the actual network functionalities, and calculating the optimal timer through our analytical model...). Both examples show a typical packet processing task. The usual DPDK implementation is shown in Listing 3.1, while our novel proposal is depicted in Listing 3.2. While both solutions include a set of Rx queues (`queue[]`) to be processed, in Listing 3.1 each thread has assigned a specific queue in an exclusive way (line 1), while in Listing 3.2 queues are shared among multiple threads and therefore require access through the `trylock()` mechanism (see line 4). In Listing 3.1 the thread tries to retrieve a burst of packets (line 4) of maximum size `BURST_SIZE`, processes it (line 7) and **immediately** scans again its set of queues, regardless of the fact that those queues may be experiencing low traffic (or no traffic at all). We highlight that this behavior is the real cause of the 100% constant CPU utilization by a single thread, as threads are working in a *traffic-unaware* manner. As for the later point, this level of CPU usage is negatively reflected on energy consumption and also on turbo-boost waste.

Listing 3.2 shows our novel approach: once the lock for a certain queue is acquired,

the thread processes that queue until it becomes empty (`while()` loop in lines 10-11), then it releases the lock (line 12) and goes to sleep for a `timeout_short` period. If a certain lock can't be granted, that queue is skipped as a different thread is already processing it: the thread changes its `curr_queue`, extracting it randomly from the set of all available queues (line 5), and goes to sleep for a `timeout_long` period.

Despite the simplicity of these examples, we believe they clearly point out the difference between a *traffic-aware* policy and a static one simply based on greedy resource usage.

Chapter 4

Metronome Adaptive tuning

In this Chapter we provide an approach to adaptively tune the behavior of the Metronome architecture. Metronome is designed to operate via a sequence of *renewal cycles* $\Theta(i)$, which alternate *Vacation Periods* with *Busy Periods*. As shown in Figure 4.1, a *vacation period* $V(i)$ is a time interval where all the deployed packet-retrieval threads are set to *sleep mode*, hence incoming packets, labeled as $N_V(i)$ in the figure, get temporarily accumulated in the receive buffer. For simplicity, we first consider a single Rx queue, then we expand our model to multiple queues in Section 4.5. When the first among the sleeping threads wakes up and wins the race, via a successful `trylock()`, for handling the incoming packets from the Rx queue, a *busy period* $B(i)$ starts. This period will last until the whole queue is depleted by either the $N_V(i)$ formerly accumulated packets, as well as the new $N_B(i)$ packets arriving along the busy period itself $B(i)$—see the example in Figure 4.1.

After depleting the queue, the involved thread will return to sleep. Note that other concurrent threads which wake up during a busy period will have no effect on packet processing—failing in the `trylock()` they will just note that Rx queue unloading is already in progress and will therefore instantly return to sleep, thus freeing CPU resources for other tasks.

4.1 Metronome Multi-Threading Strategy

As later demonstrated in Section 5.6, Metronome relies on multiple threads to guarantee increased robustness against CPU-reschedule delays of each individual Metronome thread, which is no longer in sleep state—the sleep timeout has fired and the thread was brought onto the OS run-queue. Such delay can be caused by CPU-scheduling decisions made by the OS—we recall that these decisions depend on the thread workload, their relative priorities and their current binding towards CPU-cores.

In such conditions, Metronome's control of the vacation period duration is not direct, as it would be in the single-thread case by setting the relevant timer, but it is *indirect* and stochastic, as this period is the time elapsing between the end of a previous busy period and the time in which some deployed thread awakes again and acquires the role of manager of the Rx queue. The question therefore is: *how to configure the awake timeouts of the different deployed Metronome threads*?

Unfortunately, the simplest possible approach of *equal* timeouts comes along with performance drawbacks: we will demonstrate later on (see Section 5.1.3) that when timeouts are all set to a same value, CPU consumption significantly degrades as load increases, which is antithetic with respect to the objectives of Metronome. Indeed, especially under heavy packet arrival rate, threads would wake up, therefore consuming CPU cycles, just to find out that another thread is already doing the job of unloading the Rx queue.

We thus propose a ***diversity-based*** strategy for configuring the wake up timeouts of different threads, which aims at mimicking a classical *primary/backup* approach, but without any explicit (and necessarily adding some extra CPU consumption) coordination, i.e. by using purely random access means. Each thread independently

classifies itself as being in *primary* or *backup* state, according to the following rules:

- A thread becomes *primary* when it gets involved in a service time (it is the winner of the `trylock()` based race); at the end of the busy period it carried out, it reschedules its next wake-up time after a "short" time interval T_S;
- A thread classifies itself as *backup* when it wakes up and finds an on-going busy period (i.e. another thread is already unloading the queue); it then schedules its next wake-up time after a "long" time interval $T_L > T_S$.

In high load conditions, the above rules yield a scenario in which one thread at a time (randomly changing in the long term—-see Figure 4.2) is in charge to poll the Rx queue at a reasonable frequency, whereas all the remaining ones occasionally wake up just for fall-back acquisition of the ownership on the Rx queue if for some reason the thread that was primary gets delayed, e.g. by the OS CPU-scheduling choices. Conversely, at low loads more threads will happen to be simultaneously in the *primary* state, thus permitting to significantly relax the requirements on the "short" awake timeout T_S and motivating the adaptive strategy introduced in Section 4.4.

4.2 Metronome Analysis

4.2.1 Background

Let us non-restrictively assume that, once a thread wakes up, the packets accumulated in the Rx buffer get retrieved at a constant rate μ packets/seconds.It readily follows that the duration of the busy period $B(i)$ depends on the number of accumulated packets, and, more precisely, it comprises two components: i) the time needed to deplete the first $N_V(i)$ packets arrived during $V(i)$, plus ii) the extra time needed to deplete the next $N_B(i)$ packets arrived since the start of the the busy period—in

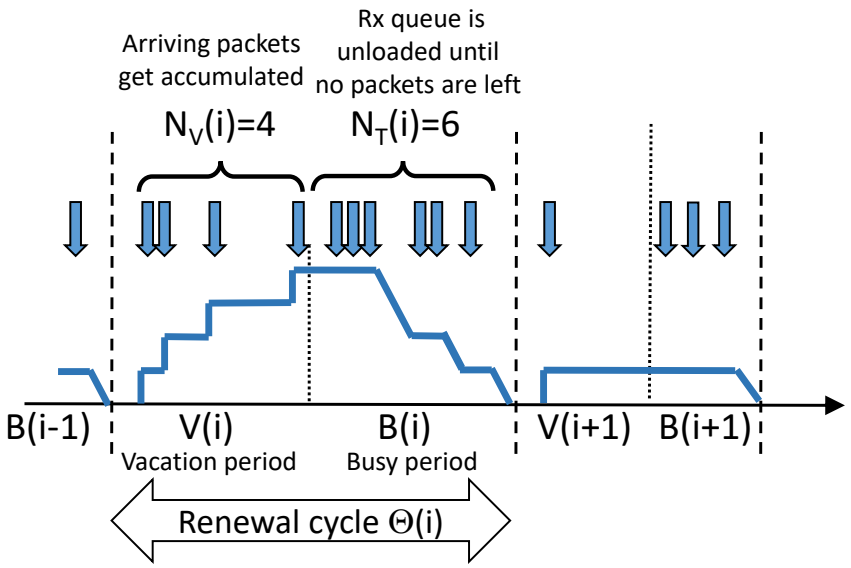


Figure 4.1: System model & renewal cycle

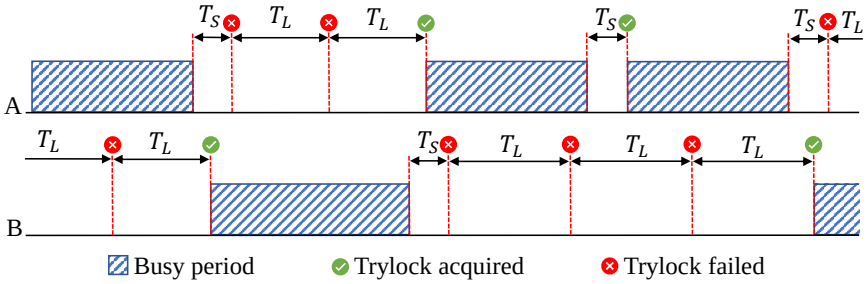


Figure 4.2: Vacation period and timeline of residual awake timeouts

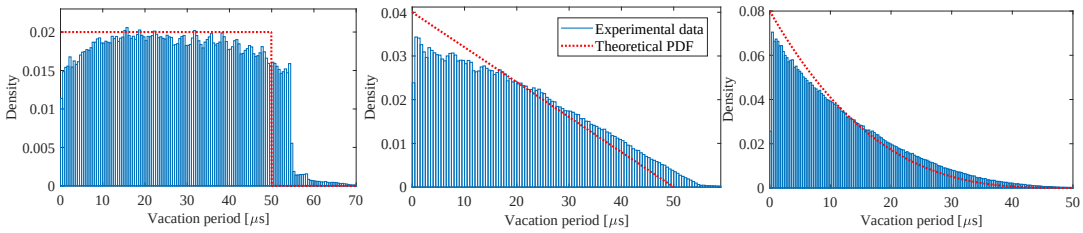
formulae:

$$B(i) = \frac{N_V(i) + N_B(i)}{\mu} \tag{4.2.1}$$

Since $N_V(i)$ and $N_B(i)$ depend on the vacation period $V(i)$, in most generality drawn from a random variable V, we can take conditional expectation at both sides of (4.2.1) with respect to V. Being λ the (unknown) mean packet arrival rate, we obtain the following fixed point equation[1] in $E[B|V]$:

$$E[B|V] = \frac{1}{\mu} E[N_V(i) + N_B(i)|V] = \frac{\lambda}{\mu}(V + E[B|V]). \tag{4.2.2}$$

[1]In the derivation, we exploited the following well known fact (direct consequence of the Little's Result): the average number $E[N]$ of packets arriving during a time interval of mean length $E[T]$ is $E[N] = \lambda E[T]$.



(a) 2 cores (b) 3 cores (c) 5 cores

Figure 4.3: Vacation period PDF: analysis vs experiments, $T_S = T_L$

which yields an explicit expression of how a busy period $E[B|V]$ is affected by the relevant vacation period:

$$E[B|V] = V\frac{\lambda/\mu}{1-\lambda/\mu} \tag{4.2.3}$$

If we conveniently define $\rho = \lambda/\mu$, we can derive an explicit expression which relates ρ to the controllable Vacation Period duration V and the relevant observable Busy Period $E[B|V]$— this expression will be indeed used to estimate ρ in Section 4.4:

$$\rho = \frac{E[B|V]}{V + E[B|V]} \tag{4.2.4}$$

4.2.2 Vacation Period statistics at high load

It is useful to start from two simplified mean-value analyses relying on two opposite sets of assumptions valid at either high load or low load. The two different models will be then blended into a single one in Section 4.4. Let $M \geq 2$ be the number of deployed Metronome threads. In *high load conditions*, for reasons that will soon become evident, we can assume that only one of such threads is in the primary state, whereas all the remaining $M-1$ are in backup state. Once the primary thread releases the Rx queue lock and schedules its short timer T_S, two possible cases may occur:

- no backup thread wakes up during the sleep timeout T_S; in this case the primary thread will get back control of the Rx queue for the next round, and will remain

primary;

- one of the remaining $M-1$ backup threads wake up *before* the end of the sleep timeout T_S and thus becomes primary; when the former primary thread wakes up, it will find a busy period[2] and will therefore acquire the role of backup thread, rescheduling its next wake up timeout after a time T_L.

Let us now make the assumption that the (current) $M-1$ backup threads were earlier CPU-rescheduled at independent random times. This *Decorrelation* assumption, indeed later on verified in Figure 4.3 using experimental results, is justified by the fact that each service time, due to its random duration, de-synchronizes the primary thread CPU-reschedule from the remaining ones; since after a few busy cycles all threads will have the chance to become primary, even if initially being CPU-scheduled at around the same times, their CPU-rescheduling instants will rapidly "decorrelate".

The statistics of the random variable V (vacation period) can be computed as the *minimum* between i) the fixed wake-up timeout T_S of the primary thread, and ii) the wake-up timeout of any of the remaining $M-1$ threads, which, owing to the previous decorrelation assumption, have been CPU-rescheduled in any random instant in the range between 0 and T_L before the end of the current busy period. It readily follows that the cumulative probability distribution function of V is given by:

$$CDF_V(x) = P(V \leq x) = \begin{cases} 1 - \left(1 - \frac{x}{T_L}\right)^{M-1} & x < T_S \\ 1 & x \geq T_S \end{cases} \tag{4.2.5}$$

and the mean vacation period for a given configuration of the short and long awake timeouts, and for a given number of threads, is trivially computed as:

$$E[V] = \int_0^{T_S} (1 - CDF_V(x))dx = \frac{T_L}{M}\left(1 - \left(1 - \frac{T_S}{T_L}\right)^M\right) \tag{4.2.6}$$

[2]In high load conditions, owing to equation 4.2.3, the average busy period lasts significantly longer than the vacation period.

Finally, the probability that one of the $M-1$ backup threads gains access to the Rx queue at its wake-up time is given by:

$$P_{s,succ} = \int_0^{T_S} \frac{1}{T_L}\left(1-\frac{x}{T_L}\right)^{M-2} dx = \frac{\left(1-\frac{T_S}{T_L}\right)^{M-1}}{M-1} \tag{4.2.7}$$

4.2.3 Vacation period statistics at low load

While, at high load, a neat pattern emerges in terms of one single primary thread at any time, with multiple backup threads, it is interesting to note that at low load Metronome yields a completely different behavior. Indeed, owing to equation (4.2.3), as the offered load reduces, the average busy period duration becomes small with respect to the vacation period duration. It follows that when a primary thread gets control of the Rx queue, it very rapidly releases such control, so that another thread waking up will find the queue available with high probability. It follows that in the extreme case, *all* threads will always remain in the primary state[3] and thus will periodically reschedule their next wake-up times after a short interval T_S. This case is even simpler to analyze than the previous one, as the CDF of the vacation time directly follows from (4.2.5) by simply setting $T_L = T_S$ and by considering M "competitors", in formulae:

$$CDF_V(x) = P(V \leq x) = 1 - \left(1-\frac{x}{T_S}\right)^M \tag{4.2.8}$$

and mean vacation period simplifying to $E[V] = T_S/M$.

4.2.4 Experimental verification of the decorrelation assumption

To verify the validity of the decorrelation assumption used in the above models, Figure 4.3 compares the probability distribution function obtained by taking derivative of the

[3]This is because each time an awaken thread finds the Rx queue not locked by another thread, then it acquires the primary role thanks to its successful `trylock()` operation.

CDF in equation (4.2.5), i.e., for $x < T_S$,

$$PDF_V(x) = \frac{M-1}{T_L}\Big(1 - \frac{x}{T_L}\Big)^{M-2} \tag{4.2.9}$$

with experimental results. We have specifically focused on the case $T_L = T_S$ as in this case the formula in equation (4.2.5) is expected to hold independently of the load (primary and backup threads use the same awake timeouts). Results, obtained with awake timeouts set to $50\mu s$ and different numbers of threads M, suggest that the decorrelation approximation is more than reasonable and the proposed model is quite accurate. Furthermore, the results also show that, in the real case—although rarely—actual CPU-reschedules after a sleep period can occur after the maximum time delay T_L, because of CPU-scheduling decisions by the OS—for example favoring OS-kernel daemons. However, such impact becomes almost negligible in Metronome with just $M = 3$ deployed threads, pointing to the relevance of the adopted multi-threading approach.

4.3 Adaptation policy under general load conditions

We propose a simplified, but still theoretically motivated, approach which allows us to blend the results obtained via the two extreme low and high load models into a single and convenient analytical framework.

More specifically, in *intermediate load conditions* we cannot anymore assume that just one single thread (as in high load conditions), or all threads (as in low load conditions), are in primary state along time. Rather, apart from the *single* thread that has last depleted the Rx queue, which is therefore surely in primary state, also *some* of the remaining $M - 1$ threads will be in primary state whereas others will be

in backup state. Let us therefore introduce a random variable P which represents the number of the remaining threads in primary state. $M-1-P$ will therefore be the number of remaining threads in secondary state.

Let us now assume that each of the remaining $M-1$ threads can be *independently* found in primary or backup state with probability p (which will be determined later on). Then, the random variable P representing the number of remaining threads in primary state trivially follows the Binomial distribution:

$$\text{Prob}(P=k) = \binom{M-1}{k} p^k (1-p)^{M-1-k}$$

Then, we can compute the average vacation time also in intermediate load conditions, by taking conditional expectation over this newly defined random variable P. This permits us to generalize equation (4.2.6) as follows:

$$\begin{aligned} E[V] &= E[E[V|P]] = \\ &= \sum_{k=0}^{M-1} \binom{M-1}{k} p^k (1-p)^{M-1-k} \int_0^{T_S} \Big(1-\frac{x}{T_S}\Big)^k \cdot \\ &\quad \cdot \Big(1-\frac{x}{T_L}\Big)^{M-1-k} dx = \\ &= \int_0^{T_S} \left(1 - \frac{px}{T_S} - \frac{(1-p)x}{T_L}\right)^{M-1} dx = \\ &= \frac{1-((1-p)(1-T_S/T_L))^M}{M\left(\frac{p}{T_L}+\frac{1-p}{T_S}\right)} \end{aligned}$$

Furthermore, assuming $T_L >> T_S$, we can conveniently simplify the above expression and approximate it as:

$$E[V] = \frac{T_S}{M} \cdot \frac{1-(1-p)^M}{p} \tag{4.3.1}$$

Note that, for $p \to 0$, namely when the probability to find another thread in the primary state becomes zero (high load conditions), equation (4.3.1) converges to the

expected value T_S, whereas $E[V] = T_S/M$ for $p = 1$ (as for low load conditions, i.e. all the threads becoming primary).

As a last step, it suffices to relate p with the offered load. To this purpose, let $\rho = \lambda/\mu$ be the probability that the Rx queue is busy at a random sample instant. It is intuitive to set $p = (1-\rho)$, as the probability p that a thread is in the primary state is the probability that when this thread has last sampled the queue, it has found it idle, i.e. $1 - \rho$. This finally permits us to formally support our proposed formula (4.4.3) as the load-adaptive T_S setting strategy. Summarizing for the reader's convenience, being $\bar{V}$ a constant target vacation period, and ρ the current load estimate, T_S can be set as:

$$T_S = M\frac{1-\rho}{1-\rho^M} \cdot \bar{V}$$

Note that this rule can be conveniently rewritten in a more intuitive and simpler to compute form, as:

$$T_S = \bar{V}\frac{M}{1+\rho+\cdots+\rho^{M-1}}$$

4.4 Metronome Adaptation and Tradeoffs

Whenever the mean arrival rate is non-stationary, but varies at a time scale reasonably longer than the cycle time, the load conditions can be trivially run-time estimated using equation (4.2.4). For instance, the simplest possible approach is to consider for $\rho(i) = \lambda(i)/\mu$ the exponentially weighted estimator:

$$\rho(i) = (1-\alpha)\rho(i-1) + \alpha\frac{B(i)}{V(i)+B(i)} \tag{4.4.1}$$

Established that measuring the load is not a concern for Metronome, a more interesting question is to devise a mechanism which adapts the awake timeouts to the time-varying load. The obvious emerging trade-off consists in trading the polling

frequency, namely the frequency at which threads wake up, with the duration of the vacation period which directly affects the packet latency. Indeed, if we assume that the serving thread is capable to drain packets from the Rx queue at a rate μ greater than or equal to the link rate, namely the maximum rate at which packets may arrive (in our single-queue experiments, 10 gigabit/s), then once the thread starts the service, packets will no longer accumulate delay. Therefore, the worst case latency occurs when a packet arrives right after the end of the last service period, and is delayed for an entire vacation period.

It follows that an adaptation strategy that *targets a constant vacation period duration irrespective of the load* appears to be a quite natural approach. Let us recall that, under the assumption $T_L >> T_S$, the average vacation period at high load given by equation (4.2.6) simplifies to $E[V] \approx T_S$. Conversely, at low load, we obtained $E[V] = T_S/M$. Therefore, being $\bar{V}$ our target constant vacation period, the rule to set the timer T_S at either high or low loads reduces to:

$$\begin{cases} T_S = \bar{V} & \text{highload} \\ T_S = M \cdot \bar{V} & \text{lowload} \end{cases} \tag{4.4.2}$$

The analysis of the general case (intermediate load) is less straightforward, but can be still formally dealt with by assuming that threads are independent and are in primary or backup state according to the probability that, while they wake up, they find the Rx queue idle or busy, respectively. As shown in Section 4.3, we can prove that, in this general case, under the assumption $T_L >> T_S$, the rule to set the timer T_S becomes:

$$T_S = M \frac{1-\rho}{1-\rho^M} \cdot \bar{V} \tag{4.4.3}$$

which, as expected, converges to (4.4.2) for the extreme high load case $\rho \to 1$ and the extreme low load case $\rho \to 0$.

Finally, we stress that Metronome does *not* sacrifice latency, but provides the *possibility to trade latency for CPU consumption.* Indeed, the duration of the chosen vacation period will determine the performance/efficiency trade-off: the longer the chosen vacation time, the lower the polling rate and thus the CPU consumption, at the price of a higher latency. If a deployment must guarantee low latency then it should either configure a small vacation time target, or even disable Metronome and use standard DPDK.

4.5 The multiqueue case

When Metronome is used with 40+Gb NICs, one queue becomes not enough to sustain line rate traffic. Therefore, a split of the incoming traffic into multiple receive queues through RSS is needed. We now introduce the N parameter, which represents the number of Rx queues for a certain NIC. Given M as the total number of threads in the system, we believe it should be at least as big as N, so that every queue can have one primary thread associated to it ($M \geq N$). In this scenario, we have N primary threads (since everyone of them has won the lock race for a different queue) and $M - N$ secondary threads. In a scenario with multiple queues, we believe it is not efficient to statically bind a thread to a certain queue (see Section 5.8.6), so we propose a different approach:

- once a **primary** thread has won the race and depleted a queue, it goes to sleep for a T_S period and when it wakes up, it contends for the same queue as we know it is likely for it to win the race again.
- once a **backup** thread has lost a lock race, it chooses the queue to be contended at its next wakeup randomly.

The random selection of the next queue for the backup thread ensures a certain decorrelation among the threads in the next queue selection and also fairness with respect to the queue checks. While the T_L value remains fixed, we update equation (4.4.3) as follows:

$$T_S = \frac{M}{N} \cdot \frac{1 - \rho_i}{1 - \rho_i^{\frac{M}{N}}} \cdot \bar{V} \quad \text{for } i = 1, \ldots, N \tag{4.5.1}$$

We notice two differences with the single queue version. The former is that the M parameter is now replaced with M/N, as that is the average number of threads taking care of a certain queue at any moment. The latter is that the ρ parameter is now per-queue based, as each queue can experiment different traffic rates (and therefore, queue occupancy and vacation periods) at any time.

Chapter 5

Experimental results

Our experimental campaign starts with the appropriate tuning for the $\bar{V}$, M and T_L parameters and the analysis of the subsequent tradeoffs. Section 5.2 shows how to convert $\bar{V}$ in terms of latency. Section 5.3 shows how Metronome adapts to a variable traffic load. Section 5.4 discusses in detail both strengths and weaknesses of Metronome and static DPDK in different aspects (latency, CPU usage and power consumption). Section 5.5 compares Metronome and XDP, while Section 5.6 shows the impact of Metronome in common CPU sharing scenarios and Section 5.7 shows how interference impacts the vacation period. While tests up to Section 5.6 have been conducted with a single Rx queue using Intel X520 NICs (10Gbps), Section 5.8 evaluates Metronome in a multi-queue scenario with Intel XL710s (40Gbps), and subection 5.8.7 evaluates a typical hundred-gigabit and beyond scenario with many queues using Mellanox ConnectX-5 (100Gbps). For evaluating the system we used a server running Linux kernel 5.4 equipped with Intel® Xeon® Silver @2.1 GHz, running the `l3fwd` DPDK application [37] on an isolated NUMA node and generating constant bit rate UDP traffic with MoonGen [38]. For benchmarking our system, we used the evaluation suite provided by Zhang et al. in [39], as well as the Intel RAPL package [40] and the `getrusage()` syscall to retrieve energy usage and CPU

consumption. Tests are done with 64B packets, as this is the worst case scenario[1]. Unless explicitly stated, the tests are executed using the `performance` CPU power governor and with parameters $\bar{V}$= 10 μs, T_L= 500 μs, M=3—each choice is motivated in the following section. Further tests for two different applications are also shown in Section 5.9.

5.1 Parameters Tuning

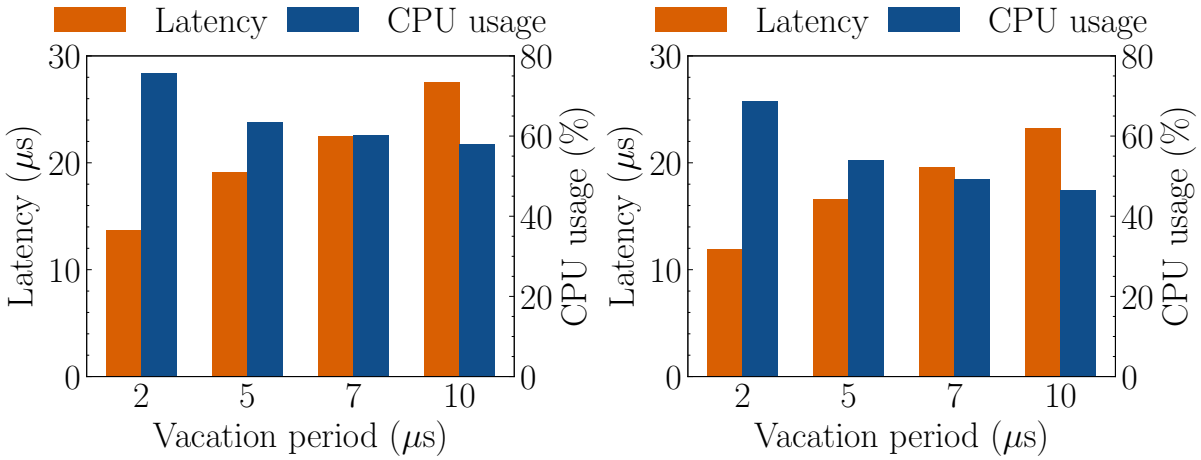
5.1.1 Vacation period ($\bar{V}$)

First of all, we would like to find a vacation period $\bar{V}$ which permits us not to lose packets under line-rate conditions. Table 5.1 shows packet loss, vacation period and busy period for different values of $\bar{V}$, which represents the target V to be used when calling the `hr_sleep()` service: we found out that 10 μs is a good starting point as it provides no loss. The test was conducted using the suite's unidirectional p2p throughput test, as this test instantly increases the incoming rate from 0 to 14.88 Mpps, so as to be sure that this setting works even in the worst case scenario. We then analyzed the bidirectional throughput scenario by assigning 3 different threads to each Rx queue, as we found out that Metronome achieves the same maximum bidirectional throughput that DPDK can reach (11.61 Mpps per port) by constantly polling each Rx queue with a different thread. Once a good suitable minimum value for $\bar{V}$ is found, we investigate how tuning $\bar{V}$ affects CPU usage and latency: indeed, as Table 5.1 shows, the shorter $\bar{V}$, the less the queue is left unprocessed as the actual (namely, the measured) vacation time V decreases, so packets tend to experience a shorter queuing period. However, such an advantage does not come for free, as

[1]For tests regarding latency, since [39] uses Moongen's timestamping capabilities, it is necessary to add a 20B timestamp to the timestamped subset of packets, thus giving rise to a minimal difference in terms of offered throughput.

Target V [μs]	Measured V [μs]	Measured B [μs]	N_V	Loss (‰)
5	11.67	13.40	172.39	0
10	19.55	20.24	287.77	0
12	21.99	22.86	326.30	0.0037
15	26.23	27.25	385.18	0.023
20	33.28	38.32	494.39	1.180

Table 5.1: Mean busy and vacation period, N_V and packet loss for different target vacation periods.



(a) 10Gbps traffic

(b) 5Gbps traffic

Figure 5.1: Latency and CPU usage for different target vacant times.

the CPU usage proportionally increases, as shown in Figure 5.1 for different traffic volumes. We note that all these tests have been performed by relying on 3 Metronome threads.

5.1.2 Number of threads (M)

As for M, the philosophy underlying Metronome is the one of exploiting multiple threads for managing a Rx queue, not the one using excessive (hence useless) thread-level parallelism. In fact, an excessive number of threads comes at almost no usefulness: Figure 5.2 shows how the percentage of busy tries increases linearly with the number of threads, along with a slight cost increase in terms of CPU usage. Further-

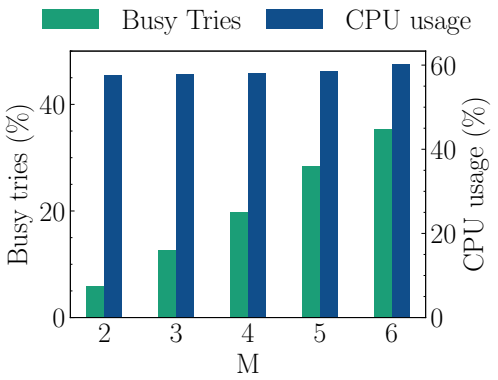


Figure 5.2: Busy tries and CPU usage versus M.

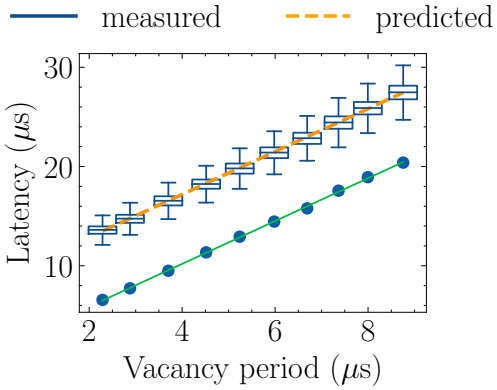
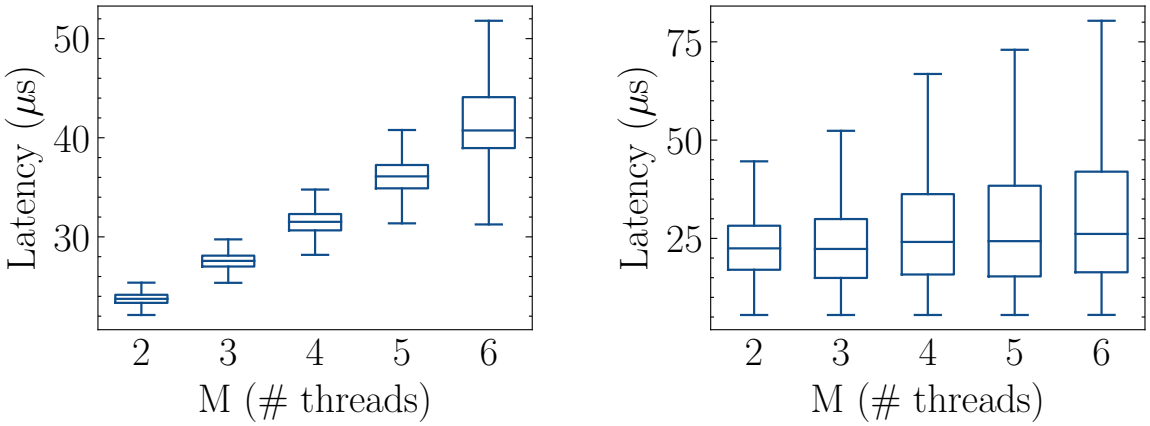


Figure 5.3: End-to-end latency (boxplots) vs. Metronome's predicted latency

more, increasing the threads number comes along with a significant cost in terms of latency, as the more the threads, the more frequently a primary thread switches to the backup role leading to longer sleep periods as stated in equation (4.4.3). We experimented considerable latency implications especially at high rates, as Figure 5.4a shows. Even for much lower rates, a substantial increase in variance is still visible (see Figure 5.4b). By the above hints, the single-queue evaluation is done with 3 threads.



(a) 10Gbps traffic

(b) 1Gbps traffic

Figure 5.4: Latency vs. the number of threads M

5.1.3 Long sleep time (T_L)

As for T_L, while letting backup threads sleep for a longer period of time alleviates the percentage of failed attempts of `trylock()` (busy tries), and therefore the number of wasted CPU cycles (as Figure 5.5 shows), a shorter T_L means higher reactivity when the primary thread is interfered by OS CPU-scheduling choices. For our evaluation, we chose 500 μs since (i) it is 50 times bigger than the maximum T_S possible value, we recall that our analytical model assumes that $T_L >> T_S$, (ii) Figure 5.5 shows that most of the advantage of increasing T_L happens before 500 μs, while between 500 and 700 μs we experimented a difference of only 1% in CPU usage and around 2% in busy tries.

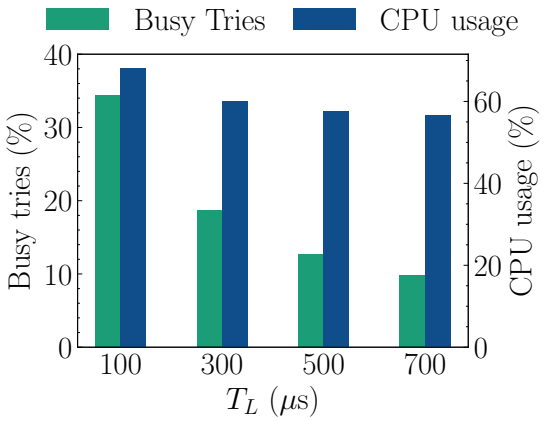


Figure 5.5: Busy tries and CPU usage versus T_L.

5.2 Tuning for latency

The adaptation strategy introduced in Section 4.4 uses, as *observable* (measurable) tuning parameter, the vacation period $\bar{V}$. It is instructive to revisit the previous analysis so as to tune the system for a desired mean latency requirement. Being N the number of packets in our system, it follows that (using the same notation and

arguments of Section 4.2)

$$E[N] = E[N_V] + E[N_B] = \lambda E[V] + \rho E[N] \rightarrow E[N] = \frac{\lambda E[V]}{1-\rho}$$

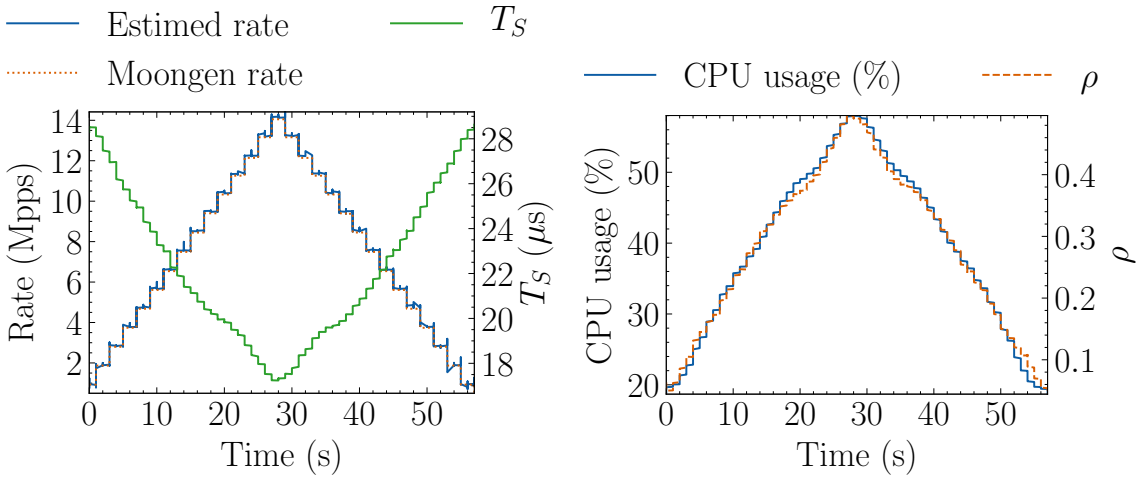
Therefore, from Little's result, we obtain:

$$E[T] = \frac{E[V]}{1-\rho} \tag{5.2.1}$$

Since a direct measure of "just" the queueing delay is technically cumbersome, Figure 5.3 compares the measured *end-to-end* latency (box plots) with the results of the above formula (green line), for various measured mean vacation periods $E[V]$ (in turns obtained with different load values ρ). Apart from the constant offset due to the propagation delay (orange dashed line) which is not included in the analytical results, these do closely match experimental ones in terms of slope of the resulting plots.

5.3 Adaptation

To test the dynamic capabilities of Metronome to adapt to varying workloads, we modified the Moongen `rate-control-methods.lua` example to generate constant bit rate traffic at a variable speed: in a time interval of one minute, Moongen increases the sending rate every 2 seconds until 14 Mpps of rate is reached at about 30 seconds, and then it starts decreasing. Figure 5.6a shows how Metronome *perfectly* matches the Moongen generated traffic rate and how the T_S parameter—set by the threads proportionally—adapts. Figure 5.6b proves that Metronome promptly adapts CPU usage with respect to the incoming traffic, starting from about 20% with no traffic and increasing up to 60% under almost line rate conditions. Also the ρ parameter correctly adjusts its value along with the traffic load.



(a) Rate and T_S estimation

(b) CPU usage and ρ

Figure 5.6: Metronome's correct adaptation to the incoming traffic load

5.4 Comparing Metronome and DPDK

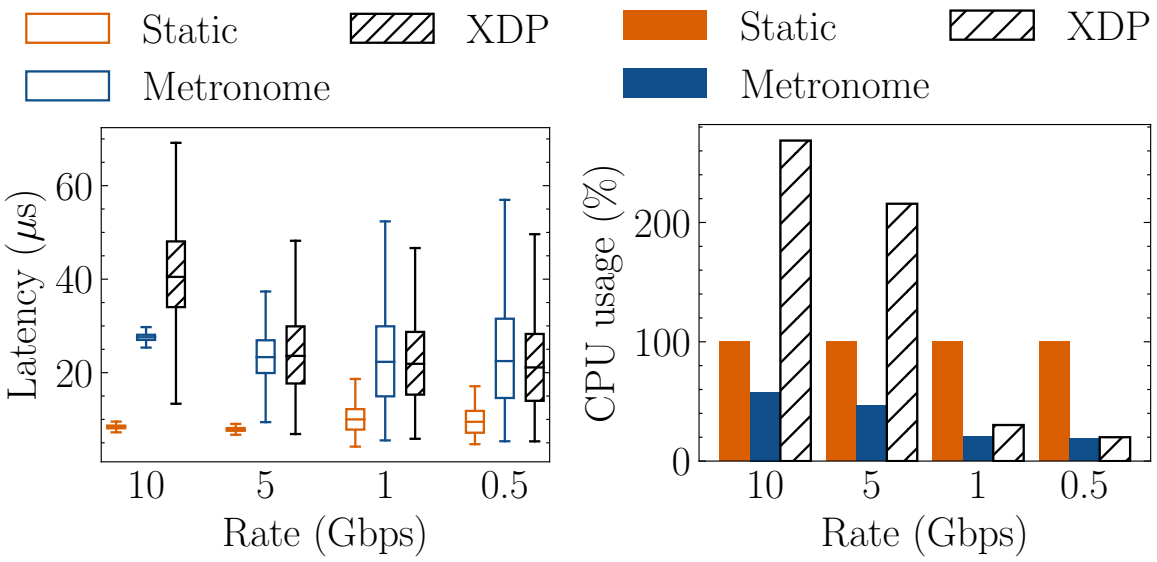
We now focus on the comparison between the adaptive Metronome capabilities and the static, continuous polling mode of DPDK in terms of (i) induced latency, (ii) overall CPU usage, and (iii) power consumption.

Latency: we tested Metronome in order to investigate how the sleep&wake approach impacts the end-to-end latency. One of our goals was to experiment a constant vacation period, therefore a constant mean latency. Figure 5.7a shows how Metronome (blue boxplots) successfully fulfills this requirement, despite a negligible increase under line-rate conditions, which seems obvious. DPDK clearly benefits from its continuous polling operations as it induces about half of the mean latency that Metronome achieves and is also more reliable in terms of variance (see Figure 5.7a - orange boxplots). However, rather than very low latency, Metronome targets an adaptive and fair usage of CPU resources with respect to the actual traffic. The minimum latency that Metronome can induce is mainly limited by two aspects: the first one is the Tx

batch parameter. Since DPDK transmits packets in a minimum batch number which is tunable, as our system periodically experiments a vacation period some packets may remain in the transmission buffer for a long period of time without actually being sent: this is clearly visible as variance at low rates increases. To overcome such a limitation, we ran another set of tests with the transmission batch set to 1, so that no packets can be left in the Tx buffer. We found out positive impacts on both variance and (slightly) mean values for very low rates. Downgrading the Tx threshold to 1 comes at the cost of a 2-3% increase in CPU utilization at line rate. The second aspect is the minimum granularity that `hr_sleep()` can support, even if the sleep time requested is much smaller than microseconds (i.e., some nanoseconds). By tuning the first parameter and patching `hr_sleep()` in order to immediately return control if a sub-microsecond sleep timeout is requested, we managed to obtain a 7.21 μs mean delay in Metronome which is very close to the DPDK minimum one (6.83 μs), and also a significant decrease in variance (0.62 μs in Metronome vs. 0.43 μs in DPDK) while still maintaining a 10% advantage in CPU consumption.

Total CPU usage: Figure 5.7b shows the significant improvements by Metronome (blue bars): while DPDK's greedy approach (orange bars) gives rise to fixed 100% CPU utilization, Metronome's adaptive approach clearly outperforms DPDK as it is able to provide 40% CPU saving even under line-rate conditions, while under low rate conditions the gain further rises to more than 5x (Metronome achieves around 18.6% CPU usage at 0.5Gbps). We underline that Metronome's CPU consumption could be further decreased by increasing the T_L value as explained in Section 5.1.

Power consumption: as for energy efficiency, it is critical to examine the two approaches depending on the different power governors[41] available in Linux. More specifically, we concentrated on the two most performing ones, namely `ondemand` and

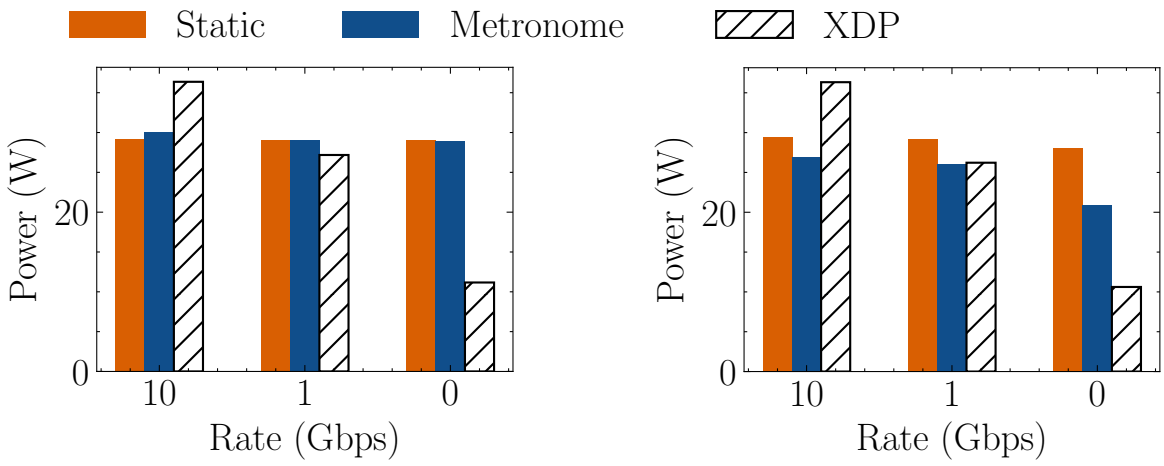


(a) Latency boxplot

(b) CPU usage

Figure 5.7: L3 Forwarder example running static DPDK, Metronome and XDP

`performance`. The first can operate at the maximum possible speed, but dynamically adapts the CPU frequency by periodically examining the current CPU load and depending on some threshold values, while the second one keeps the CPU cores at their maximum speed while executing code. While `ondemand` permits a more adaptive CPU policy, it is less reactive than `performance`. In particular, CPU cores need more time to get to the maximum speed, but this permits some savings in terms of power. This trade-off is clearly visible in Figures 5.8a and 5.8b: except for the 10Gbps throughput under the `performance` power governor scenario, Metronome achieves less power consumption than the traditional DPDK does, with the maximum gain reached when operating under no traffic with the `ondemand` governor (around 27%). We underline that in the `ondemand` scenario Metronome's CPU usage is higher than in the previously seen plots (∼90% at 10Gbps, ∼70% under no load). While we concentrated on the `performance` governor since we wanted to minimize Metronome's CPU consumption, these tests show that depending on the user/provider's needs, Metronome can



(a) `performance` (b) `ondemand`

Figure 5.8: Power vs CPU utilization for different power governors.

also achieve significant power saving when compared to static polling DPDK.

5.5 Comparing Metronome and XDP

We believe it is the case for Metronome to be also compared with XDP [31]: this work has a similar motivation to Metronome's main one (reduced, proportional CPU utilization) and is nowadays integrated into the Linux kernel. Despite this similar goal, the approach of the two architectures is quite different: XDP is based on interrupts and every Rx queue in XDP is associated to a different, unique CPU core with a 1:1 binding. Through a conversation with one of the XDP authors on GitHub [42], we discovered that our Intel X520 NICs (running the `ixgbe` driver) achieve at their best a close-to line-rate performance: in fact, the maximum we managed to get is 13.57 Mpps with 64B packets. To do this, we had to equally split flows between four different cores running the `xdp_router_ipv4` example (the most similar one to DPDK's `l3fwd`). The graphs now discussed are obtained using the minimal number

of cores for XDP in order not to lose packets[2] (4 cores on 10Gbps and 5Gbps, 1 core on 1Gbps and 0.5Gbps). We remark that if XDP is deployed with the goal of potentially sustaining line-rate performance, on our test server it should statically be deployed on four cores since there's no way to dynamically increase the number of queues (and therefore, cores) without the user's explicit command through `ethtool`: in that case, XDP's total CPU usage increases at 52% @1Gbps and 34% @0.5Gbps. Figure 5.7a shows the latency boxplot for XDP: while (even with interrupt mitigation features enabled) we see an increased latency at line rate, we experimented similar latencies at lower rates (we underline that decreasing Metronome's $\bar{V}$ and the Tx batch parameter we could obtain lower latency results as shown in Section 5.4, while XDP is already operating at its best performance). Figure 5.7 shows XDP's mean total CPU utilization, which is clearly much higher because of the per-interrupt housekeeping instructions required to lead control to the packet processing routine, which have an incidence, especially at higher packet rates. On the other hand, XDP occupies no CPU cycles at all under no traffic, while Metronome still periodically checks its Rx queues. This different approach permits Metronome to be highly reactive in case of packet burst arrivals (as shown in Section 5.1), while XDP loses some tens of thousands of packets in this case before adapting. In terms of power consumption (Figure 5.8), for the same reason discussed above XDP consumes more power when processing at line rate, while permitting significant gains when no traffic is being processed.

[2]We decreased the Mpps sending rate to 13.57 by sending 72B packets, so that XDP wasn't losing packets.

5.6 Impact

We now analyze Metronome's capabilities to work in a standard CPU-sharing scenario, where different tasks compete for the same CPU. We first focus on motivating our multi-threading approach, then we show that the CPU cycles not used by Metronome can be exploited to run other tasks in the meantime without significantly affecting Metronome's capabilities. In both the experiments, Metronome shares its same three cores with a VM running `ferret`, a CPU-intensive, image similarity search task coming from the PARSEC [43] benchmarking suite. [3]. Because the Metronome task is more time sensitive than the `ferret` one, we give Metronome a slight scheduling advantage by setting its niceness value to -20, while the VM's niceness is set to 19 since it has no particular time requirements. In any case, the two are still set to belong to the same `SCHED_OTHER` (normal) priority class.

The case for multiple threads: While we previously stated that a few threads are better for Metronome, we now clarify the reason for using multiple threads by scheduling the VM running the `ferret` program on one core. When running Metronome on the same single core, because of the CPU conflicting scenario the maximum throughput achievable by `l3fwd` is around 8 Mpps. If we deploy Metronome on three cores (one of these three cores is the same used by the VM), only one thread will be highly impacted by the CPU-intensive task and therefore will unlikely act like a primary thread. In this case `l3fwd` achieves no packet loss on a 10Gbps link, and the same scenario happens if we schedule the same VM running `ferret` on two of the three cores shared with Metronome. The next paragraph shows that also when all of the three Metronome threads are (potentially) impacted by `ferret`, they can still for-

[3] We focused on a CPU-intensive task since it is less likely to release the CPU. We have also tried Metronome with more memory-intensive PARSEC tasks such as `canneal` and found out that the results are pretty similar.

ward packets at line rate, thanks to the reduced likelihood that all of them (when requiring to be brought back to the runqueue after the sleep period) are impacted simultaneously because of the decisions of the OS CPU-scheduler. These experiments clearly show that running Metronome on multiple threads leads to improved robustness against common CPU sharing scenarios and interference by other workloads.

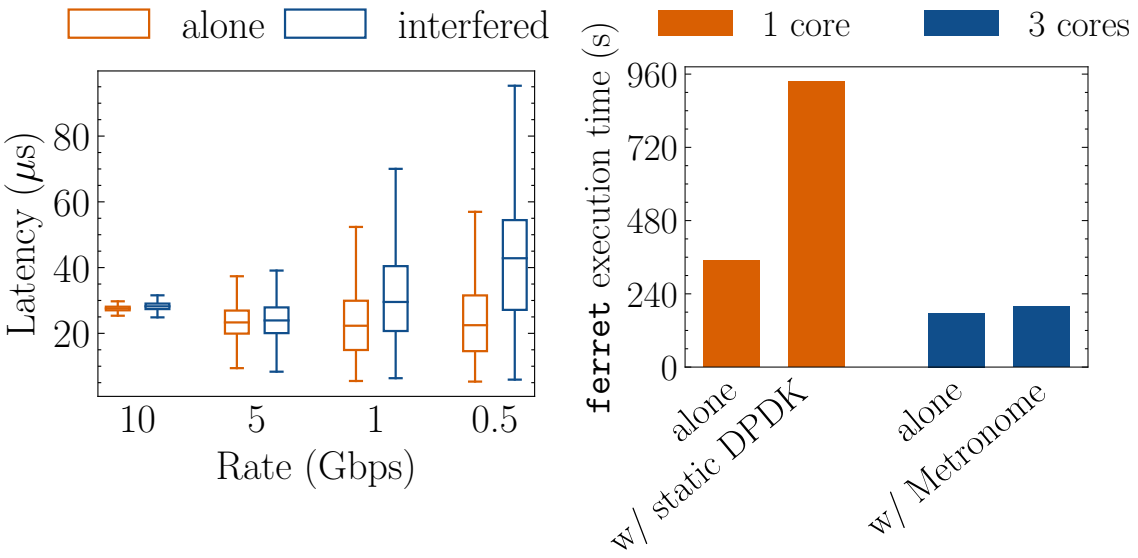
Vacation period impact: In the Appendix, we show the same test performed in Section 4.2.2, Figure 4.3, with the addition of `ferret` interfering Metronome.

Latency impact: Figure 5.9a shows the latency boxplots for Metronome with the `ferret` interference (blue) and without it (orange). The impact of interference on latency is that, once Metronome releases the CPU and consequently awakens, it will wait for some time before being rescheduled since the VM is not preempted immediately by the OS. This phenomenon of course happens less frequently under high load, as Metronome has more work to do and therefore is less likely to release the CPU, while it is more likely under low load, where the increase in latency is indeed more visible.

Co-existence with other tasks: we now demonstrate that Metronome's sleep&wake approach enables the CPU sharing of other tasks without major drawbacks, while DPDK's static, constant polling approach denies such possibilities. We first ran `ferret` on one core, with a static DPDK polling `l3fwd` application on the same core. Then, we scheduled `ferret` on three cores and the three Metronome threads on the same cores. As Figure 5.9b shows, sharing the CPU with a static polling task causes `ferret` to almost triple its duration, while Metronome's multi-threading and CPU sharing approach only causes a 10% increase. Moreover, standard DPDK's single core approach couldn't keep up with the incoming load, achieving a maximum of 7.31 Mpps, while Metronome achieved no packet loss even when all of its three

	alone	w/ `ferret`
static DPDK	14.88	7.34
Metronome	14.88	14.88

Table 5.2: Throughput (Mpps) for static DPDK and Metronome



(a) Metronome: latency boxplots

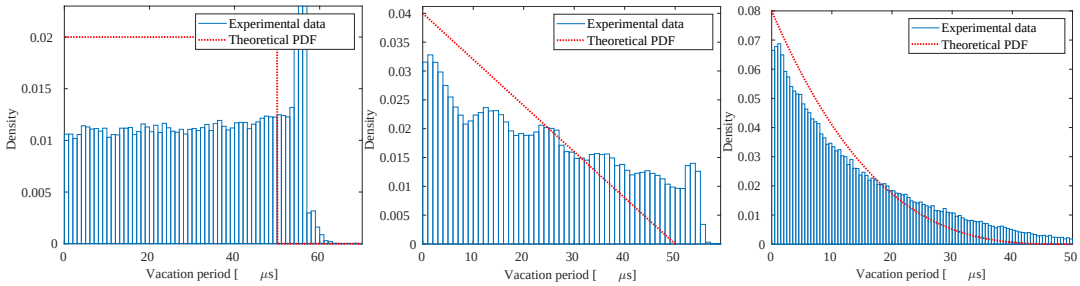
(b) `ferret`: execution time

Figure 5.9: Tests with Metronome running alone and interfered by `ferret`

cores were shared with a CPU intensive program such as `ferret` (see Table 5.2). We underline that Metronome's multi-threading strategy implies that the same workload is shared between multiple threads, thus the more the cores, the less the work every thread needs to perform and therefore the more they can co-exist with other tasks without affecting performances, as this test shows.

5.7 Vacation period interference

We present here the results in which we replicate the experiment in Section 4.2.1, Figure 4.3. The scenario here is always $T_L = T_S = 50\mu s$, with `ferret` competing for all the cores. From the experimental results, it seems like when M cores are



(a) 2 cores (b) 3 cores (c) 5 cores

Figure 5.10: Vacation period PDF: analysis vs experiments, $T_S = T_L$. Metronome is interfered by `ferret`

interfered, Metronome acts like the case of M-1 non-interfered threads. Figure 5.10a is much likely to a Dirac delta function, while 5.10b can be seen as similar to a uniform PDF. However, we believe the main takeaway from these tests, as already stated, is that the gap with the non-interfered version gets minimal as the number of threads increases (see Figure 5.10c), this further motivates our choice of using multiple threads for improved robustness.

5.8 Going multiqueue

Our evaluation now focuses on the multiqueue case analyzed in Section 4.5: tests have been conducted for both Metronome and static DPDK using Intel XL710 40Gbps NICs. These devices are limited by a maximum processing rate of 37Mpps [44]. In all tested environments, Metronome always reached the desired 37Mpps forwarding throughput. Traffic is distributed equally among the Rx queues through RSS, while in a later subsection we will discuss the unbalanced traffic case. We found out that the main components to be tuned for achieving the best performances in Metronome (assuming a fixed $\bar{V} = 15\mu s$) are the number of Rx queues, the CPU power governor and the number of threads.

5.8.1 Tuning the number of queues

We test our `l3fwd` application using 2,3 and 4 Rx queues for the same 37Mpps throughput. Results for CPU and power consumption are available in Figure 5.11. We now focus on the `performance` power governor (Figures 5.11a,b,c), as we discuss the impact of the `ondemand` power governor in the next paragraph. The ρ parameter and the busy tries percentage are also shown (see Figure 5.12) in order to better explain the results. As with 2 Rx queues every queue is experiencing high load traffic (˜18Mpps each), most of the time the queues are busy ($\rho = 0.7$ with 2 threads) and the CPUs are running at their maximum, so the main gain is in the CPU occupancy (150% with 2 threads, 156% with 8). While in the cases with many threads Metronome uses more power than static DPDK (here represented with dotted lines), it does not make much sense to use more than 4 threads to contend just two queues, as also the linear increase in busy tries (blue-filled bars in Figure 5.12a) suggests. When using a larger number of queues (3 or 4), the lower per-queue load permits Metronome to increase its gain compared to static DPDK both in CPU and power (see Figure 5.11c). In order to determine the number of queues to use, it is worth noticing that on one hand with a larger number of queues, ρ decreases and, consequently, the number of busy tries decreases, which makes the Metronome algorithm more efficient. On the other hand, as the number of queues increases, so does the number of threads to deploy and consequently, power consumption. We believe that $\rho \sim 0.5$ is a good compromise, which in our experiments is reached with 3 Rx queues.

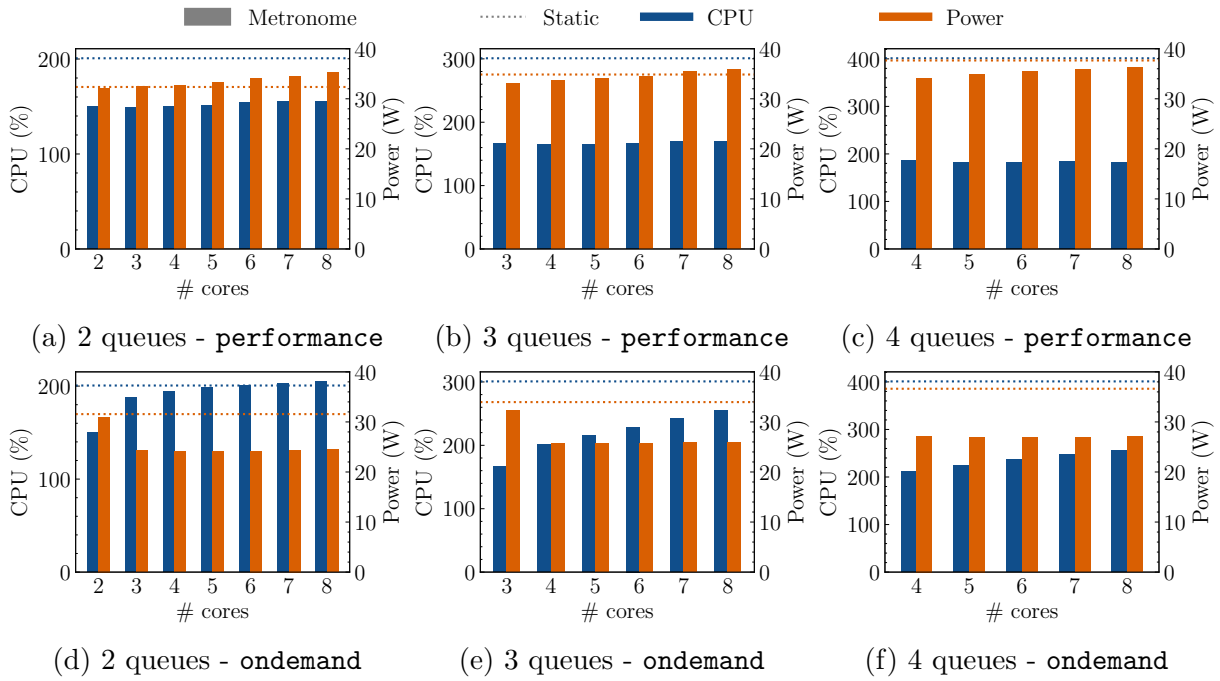
5.8.2 Power governors matter

While in the previous paragraph we focused on the `performance` governor, we now discuss the `ondemand` one, the difference between the two is explained in Section

5.4. Figure 5.11d shows the results with 2 Rx queues: the initial decrease in power consumption is motivated by the fact that while with 2 threads, these can only be in the primary state, when increasing the number of threads, they tend to be backup ones (and therefore, to sleep for more time) because of the high percentage of busy tries (see the red-filled bars in Figure 5.12a). This is in turn caused by the steep increase of ρ with the number of threads: since the CPU cores can execute at slower rates, threads will likely take more time to unload their Rx queues and therefore these will be busy for longer periods. This phenomenon is still visible with 3 queues and slightly with 4 queues. As the number of queues increases, the difference between the two power governors in terms of queue occupation ρ and busy tries still remains significant but also slightly decreases (see the subfigures in Figure 5.12). Overall, the `ondemand` power governor permits to trade some extra CPU time for a better power efficiency: also in this case the best advantages are visible with a larger number of queues. This further demonstrates Metronome's capability to adapt to a lower per-queue load.

5.8.3 Tuning the number of threads

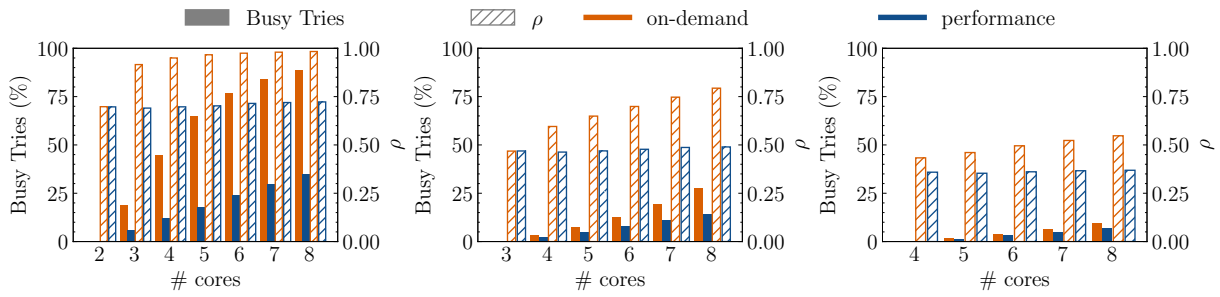
After presenting Figure 5.12, we can now draw some implications about the number of threads to be used in the multiqueue case. Our goal is to deploy a large enough number of threads (at least as big as the number of Rx queues, see Section 4.5) without incurring in too much busy tries (quantitatively, an upper bound of 10% of the tries), assuming the use of the `performance` power governor. We can see from Figure 5.12 that this is approximately achieved when the number of cores doubles the number of Rx queues. Therefore, we propose $N \leq M \leq 2N$ as a reasonable choice and we use this approach also in the later sections. In the supplementary material we show how this approach scales well also in 100Gb link scenarios.



(a) 2 queues - `performance` (b) 3 queues - `performance` (c) 4 queues - `performance`

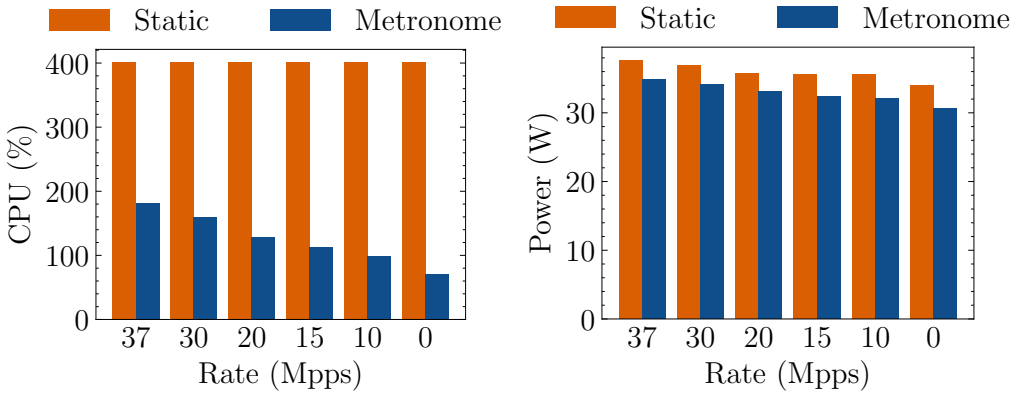
(d) 2 queues - `ondemand` (e) 3 queues - `ondemand` (f) 4 queues - `ondemand`

Figure 5.11: CPU and power for Metronome and static DPDK with different RX queues and power governors



(a) 2 queues (b) 3 queues (c) 4 queues

Figure 5.12: Busy tries and ρ with different # of Rx queues



(a) CPU usage

(b) Power

Figure 5.13: CPU and power consumption under different loads.

5.8.4 Scaling to the actual traffic

Figure 5.13a shows the CPU consumption for Metronome and DPDK under different traffic rates, from 0 to line rate on an Intel XL710 (37Mpps). The test is done with 4 Rx queues with both Metronome and DPDK, and with $M = 5$ and $\bar{V} = 15\mu s$ for Metronome. Our approach saves more than half of static DPDK's CPU cycles while maintaining the same line-rate throughput, and improving even more at lower rates. Also in terms of power consumption (see Figure 5.13b), Metronome provides around 2-3W of advantage even when using a highly expensive power governor such as `performance`.

5.8.5 Unbalanced traffic

We test Metronome's multiqueue capabilities by continuously sending at line rate an unbalanced pcap file. The file is composed by 1000 packets, 30% of the packets belongs to the same UDP flow, while the other 70% is randomly generated and therefore equally split among the queues. In the test we use 3 Rx queues (without losing packets), so the most stressed queue processes around 53% of the total throughput,

	Busy tries (%)	Total tries	ρ
Queue #1	1.94	5970660	0.3208
Queue #2	4.39	2625007	0.7269
Queue #3	2.02	5704167	0.3552

Table 5.3: Statistics for the unbalanced traffic case

while the other two queues are in charge of 23% each. Table 5.3 shows some meaningful statistics for the test. Queue #2 is the most stressed. Therefore, it has the highest busy tries percentage and also the highest queue occupation ρ. It is worth noticing that, on a 3-minute test, queue #2 experienced less than half of the lock tries of queues #1 and #3: this trend validates the assumption in Section 4.1, where a busy queue tends to have only one primary thread at a time while a less occupied one is more likely to have more threads in the primary state simultaneously, and therefore, more tries.

5.8.6 Thread-to-queue binding policy

We show that our random binding policy of a thread to a certain queue leads to increased resiliency with respect to the static binding one. We use a setup with 6 threads and 3 Rx queues: in the static binding scenario, each Rx queue is assigned to two threads in an exclusive way, while in the other case each thread chooses its next queue randomly. We interfered two threads contending the same Rx queue with `ferret`, in this case the system's throughput decreased to 27.64 Mpps, while the random policy reached line-rate throughput as always. With this test we show that Metronome's dynamic and random behavior permits increased resiliency when some of the threads are interfered, as in this case we only need (with M Rx queues) *any* of the M threads to be not interfered in order to timely process packets, while in the static binding case we need *for every* Rx queue, at least one if its threads to be not

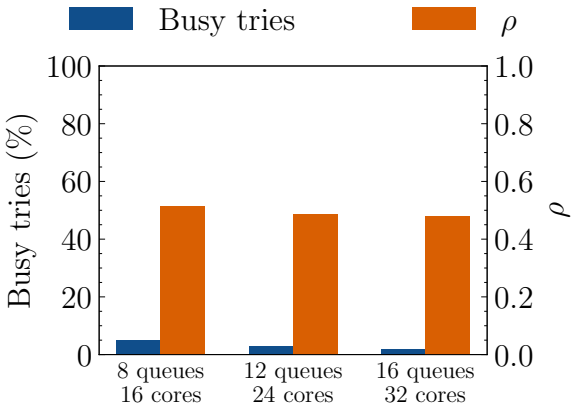


Figure 5.14: Busy tries and ρ in a many-queues scenario.

interfered.

5.8.7 Many-queues scaling

We run some preliminary tests to see if the desired goals mentioned in Section V.F ($\rho \sim 0.5$, busy tries $\leq 10\%$) are achievable also in a many-queues scenario. For this reason, we deployed Metronome using Cloudlab's Utah c6525 servers [45], which permit us to use Mellanox ConnectX-5 100Gbps NICs. We run Metronome with the $M = 2N$ parameter setting, with N=8,12,16 Rx queues. As Figure 5.14 shows, Metronome can maintain the desired $\rho \sim 0.5$, while the busy tries value decreases as the number of queues grows and the cores/queues ratio remains constant. This trend was already visible in Figure 14 of the article.

5.9 Tested applications

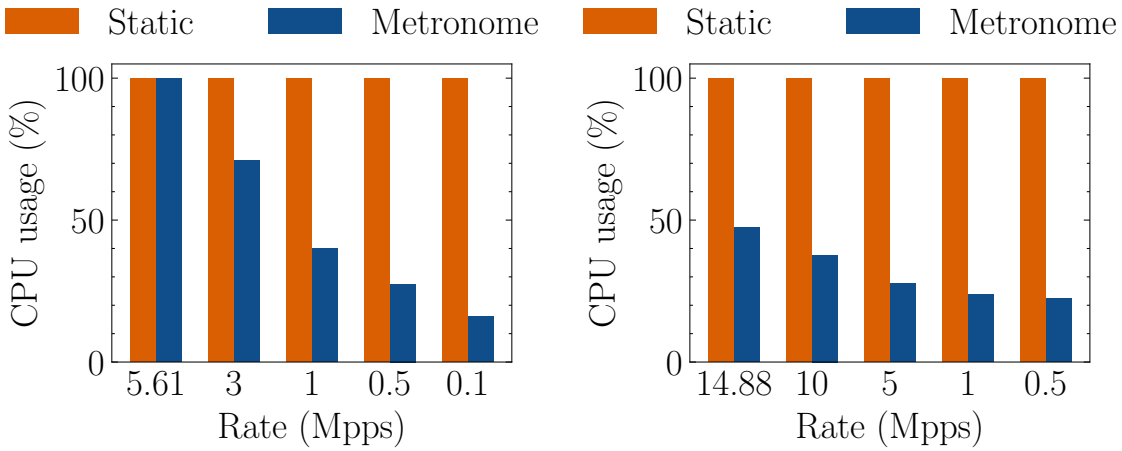
To further assess the flexibility and the wide breadth of Metronome, we show three DPDK applications that we successfully adapted to the Metronome architecture, namely two DPDK sample applications as a L3 forwarder [37] and an IPsec Security Gateway [46], as well as FloWatcher-DPDK [47], a high-speed software traffic

monitor. While these applications work at packet level, we underline that Metronome could be also integrated in frameworks communicating with upper layer applications, like mTCP [48], a TCP stack exploiting DPDK: Metronome could be integrated on the Rx side of mTCP by replacing the static polling of the NICs with our adaptive algorithm.

L3 forwarder The `l3fwd` sample application acts as a software L3 forwarder either through the longest prefix matching (LPM) mechanism or the exact match (EM) one. We chose the LPM approach as it is the most computation-expensive one between the two. We have used the `l3fwd` application to test Metronome's performances in this Chapter exhaustively.

IPsec Security Gateway This application acts as an IPsec end tunnel for both inbound and outbound network traffic. It takes advantage of the NIC offloading capabilities for cryptographic operations, while the application itself performs encapsulation and decapsulation. Our tests perform encryption of the incoming packets through the AES-CBC 128-bit algorithm as packets are later sent to the unprotected port. The DPDK sample application achieves a maximum outbound throughput of 5.61 Mpps with 64B packets in static polling mode: once we adapted the application to Metronome, we found out that we were able to reach the exact same throughput. In fact, one of Metronome's threads was always processing packets and therefore never releasing the trylock shared with the other threads, this is clearly visible in Figure 5.15a. For lower rates, Metronome clearly outperforms the static approach as rates decrease.

FloWatcher-DPDK FloWatcher is a DPDK-based traffic monitor application providing tunable and fine-grained statistics, both at packet and per-flow level. FloWatcher can either act through a run to completion model or a pipeline one: we chose the for-



(a) IPsec Security Gateway

(b) FloWatcher-DPDK

Figure 5.15: CPU usage - other applications (single RX queue)

mer since the receiving thread is also calculating the statistics, therefore providing a more challenging scenario for Metronome. We find out that Metronome provides the same performances that the static DPDK approach does in terms of zero packet loss and correct statistics calculation, while reaching major improvements in CPU utilization. In particular, Figure 5.15b shows a 50% gain even under line rate traffic and almost a 5x gain with 0.5 Mpps traffic.

Part II

Non-blocking, single-queue driver

Introduction

In this part, the author proposes a non-blocking mechanism for network drivers, which is the second part of this PhD dissertation. The main goal of this part is to design a network stack explicitly suited for a predictable, tail-bounded latency and scaling to multi-cores, thus solving Problems 3 and 4 in Section 1.2.

This approach permits the adoption of a scale-up mechanism for network stacks as opposed to the current scale-out policy (as shown in Section 2.2). By allowing more threads to process the same Rx queue in parallel, the approach enables a work-conserving property for network stacks.

We imagine two reasons why this approach has never been explored before:

1. A shared queue implies the possibility of breaking per-flow consistency, with packets in the same flow ending up (simultaneously) in different threads. It also introduces the case of packet reordering in flow-based streams;

2. Rearchitecting the stack's queue policy requires the network driver's modification. As also said in 2.1.1, users typically know very little about how drivers work.

From a pragmatic perspective, the software logic that queuing systems rely on admits a single execution flow at any time for managing the data structure implementing the queue. In particular, even if Metronome (Part I) has been shown that

multiple threads can process the packets incoming from a single queue, only one of these threads is enabled to carry out the actual operations at any time, thanks to a queue-locking mechanism.

The author believes the aforementioned motivations and the practical way to proceed with queue management—in particular, the reliance on locking and critical sections for the operations on a single queue—can be overcome for the following reasons:

1. A new generation of concurrent algorithms for handling shared data structures (e.g., [49, 50, 51, 52, 53]), which do not rely on locking mechanisms, have gained significant interest for both scalability and performance reasons. The networking community hardly knows these algorithms, and their baseline design criteria are currently not exploited at the level of queue management drivers;

2. Most of the data-center flows are restricted to a handful of packets [54], therefore minimizing both the possibility and the effects of packet reordering when adopting concurrent (e.g. simultaneous) management of the queue by multiple-threads;

3. Recently, the networking community has shown some interest in network drivers [19, 55, 56]. This has permitted the community to dive deeper into how a driver operates under the hood and how the side of actual software execution flows could further optimize it.

The author builds on these three observations and presents the first implementation (to the best of its knowledge) of a work-conserving, parallel network driver, where threads use no locking mechanism for managing the data structure, implementing a single queue. Therefore, the driver follows a scale-up policy for network stacks, which

is fully orthogonal—and mixable with—the widely adopted scale-out policy.

This part is divided into the following Chapters:

- Chapter 6 presents an algorithm fully supportable with any standard ISA, like the one offered by x86 processors, where multiple-concurrent threads can at the same time process different (sets of) packets incoming from the same queue. In this solution, thread coordination—for avoiding inconsistencies—purely occurs via the exploitation of Read-Modify-Write (RMW) machine instructions. The algorithm has been implemented on top of x86/Linux machine and has been integrated within the DPDK packet processing framework [9];

- Chapter 7 presents a comprehensive assessment of the capabilities that are permitted by this solution, compared to the classical literature scenario where every single queue is instead managed by an individual execution flow (a single thread) at any time. The scale-up mechanism shows significant improvements both in mean and tail latency, as well as minimal packet reordering as the packet size increases.

This part partially includes figures and verbatim copies of the text from the paper presenting the non-blocking driver.

Chapter 6

Architecture

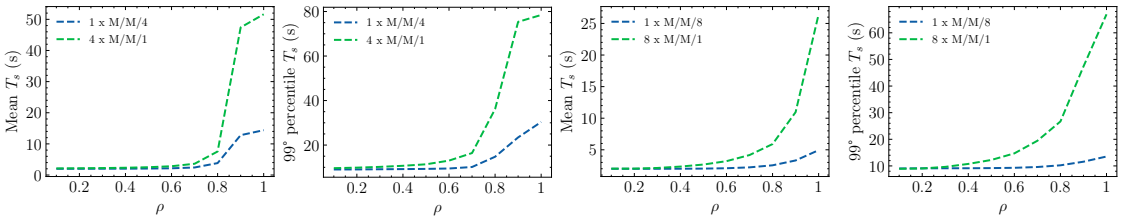
6.1 Motivation

As already explained in Section 2.2, in a modern network stack each queue is processed by only one thread, and the threads cannot simultaneously work on the same queue. We can therefore model the network stack as a $N \times M/G/1$ queuing system, where N threads have a separate queue each. It is well known from basic queuing theory (see the following subsection) that an $M/G/N$ system would bring significant advantages in tail latency; in fact, a single queue shared among the N threads enables global visibility of the workload to be processed by all the threads, therefore implying a work-conserving policy for network stacks. This is particularly beneficial in the case of head-of-the-line blocking, variations in the service time, and also in the case of temporal queue imbalances, where for short periods of time, one queue can be much populated while others may find themselves empty.

6.1.1 Simulation results

To evaluate the effectiveness of this approach from a theoretical perspective, we conducted several queuing theory simulations using the Matlab Simevents package,while varying the number of servers (4 and 8).

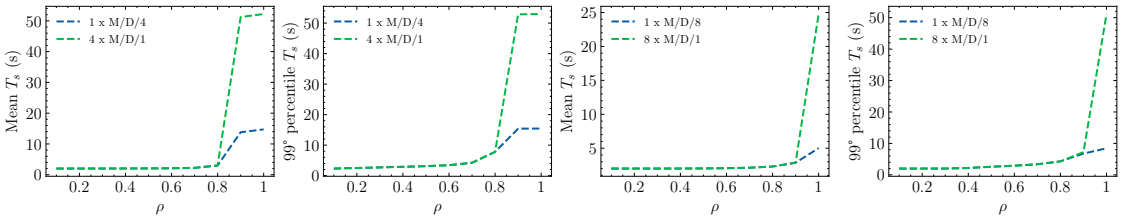
The results, presented in Figure 6.1, display both mean and 99p latency. Figure 6.1 shows results for a Markovian service time. For each plot, the blue line shows our approach, while the green one shows the current state-of-the-art. **Our findings clearly demonstrate that using multiple threads aligns with queuing theory and significantly improves both mean and tail latency.**



(a) Mean latency - 4 cores (b) 99p latency - 4 cores (c) Mean latency - 8 cores (d) 99p latency - 8 cores

Figure 6.1: Mean and 99p latency simulation results - Markovian service time

We now repeat the same test with Deterministic service times, a scenario highly unlikely to happen in modern computer architectures because of the many sources of variability. Still, this utopian scenario represents the case with fewer benefits for the proposed approach. Results are shown in Figure 6.2; it is interesting to underline that our approach still brings benefits at a very high load.



(a) Mean latency - 4 cores (b) 99p latency - 4 cores (c) Mean latency - 8 cores (d) 99p latency - 8 cores

Figure 6.2: Mean and 99p latency simulation results - Deterministic service time

In a real scenario, the core point stands in how to build the multithreaded concurrent queuing system in an effective manner—for example, via the well-suited exploita-

tion of machine instructions in the RMW class. Challenges and constraints related to these aspects are discussed in Section 6.3.

6.2 Core Concepts

This work's main idea is to design a non-blocking algorithm that can enable multi-threaded, simultaneous processing of the same Rx queue—hence avoiding any locking of the queue. From the explanation of a driver's receive routine in Section 2.1.1, it is clear that the receive flow of a network driver is anything but tailored to a concurrent execution model: in fact, the NIC-shared data structures lack the support for simultaneous operations in order to access them in parallel without causing inconsistencies (and therefore, malfunctions) in the buffer state. This causes the whole Rx function to be a critical section.

This work proposes a different approach based on the core concepts that drive the development of modern concurrent algorithms [53, 50, 57, 58, 51, 52, 59]. In these solutions, the notion of atomicity (hence correctness) of the operations by a thread is no longer linked to the concept of critical section. Rather, threads are coordinated by the reliance on atomic machine instructions belonging to the Read-Modify-Write (RMW) class. These instructions can access a memory location and update it—for example if the original value matches a target value selected by the thread.

Some of these instructions can fail in the update operation—e.g. if the memory location does not (or no longer) match the target value that has been selected. Hence, the failure makes the thread know that the shared data structure has changed its state—e.g. because of operations occurring by a concurrent thread. Threads can therefore fail/win a race in constant time, and in case of a win, the thread has earned the right to perform a specific operation, which is immediately visible to the other

threads (i.e. they fail) so that race conditions are avoided. In case of a fail, a thread has not modified the shared state and hasn't caused any delay or inconsistency for the concurrent workers. As a result of this design, the threads are totally decoupled **(a part from some corner cases occurring in our network driver, which are explained in Section 6.5.1)**, enabling total independence (they do not block each other) and resilience against slowdowns (e.g., de-scheduling, cache misses, and interrupts).

The direct consequence is the possibility of a scale-up mechanism in current network functions and end-hosts, opposite to the current scale-out policy (Figure 2.2).

6.3 Challenges and Constraints

Let us go back to Listing 2.1.1 in Section 2.1.1 to understand the portions of code where race conditions among concurrently operating threads can occur:

1. in lines 15-17, a concurrent copy and replace of the same descriptor can happen from more threads at the same time, causing inconsistencies in the shared buffer.

2. at line 23, the TAIL write is dependent on the timing of the thread's operations.

There is also a fundamental question to be investigated: for how the problem has been presented until now, the reader might think that concurrent algorithms for accessing a shared ring buffer have already been implemented in software and, therefore, may not see any significant contribution. The main difference with respect to the existing algorithms is that in the latter case, both producers and consumers are written by the user in customized software, while in this case, we can only write the consumer's portion of code without having any possibility to modify the producer's behavior, which is the NIC. Consequentially, there is also the need to make this algorithm

compatible with what the NIC is expecting, namely a single execution flow coherently processing the receive queue. This is both a requirement for running this driver with unmodified NICs and a limitation in how the threads can behave since the NIC must not notice that multiple threads are simultaneously processing the same queue.

6.4 The Algorithm

6.4.1 Handling thread-level parallelism

Let's see how we can overcome the above-mentioned conflicts through Listing 6.1:

1. **Split the work to be done:** the set of the available (in the sense of containing a packet) descriptors must be *partitioned* disjointedly among the threads, so that they don't overlap. This is achieved through the following operations: first, a scan of the Rx queue is done by reading the DD bit (lines 12-19) in order to understand how many descriptors have been filled by the NIC. At this point, the thread tries to obtain that specific batch of descriptors through an atomic Compare-And-Swap (CAS) machine instruction of the RMW class (line 21). In case of a win, the `queue->rx_index` global variable has been instantaneously updated, and therefore no other thread can obtain a conflicting set of descriptors[1]. In lines 23-30, the thread can copy the descriptors to its own buffer and replace them with new, empty ones. This is the actual portion of code we can speed up in this execution model.

2. **Synchronize on who should update the TAIL register:** we avoid concurrent TAIL writes through a trylock (lines 35 and 42). We underline that even

[1]Each conflicting thread has two scenarios: either it sees the new value of `queue->rx_index` when getting a copy at line 8 or if they still have the old one, they will fail the race for modifying it at line 21.

if the `trylock()` call fails, there are no negative consequences for the thread in terms of waiting or delay.

6.4.2 Handling transparency to the NIC

This above-mentioned NIC compatibility problem calls for a *transparency* mechanism, which is a way for making threads simultaneously process the same queue while giving the illusion to the NIC that it is interfacing with only one thread. More specifically, with *transparency* it is meant that the set of descriptors one can re-assign to the NIC *must* be contiguous; in this way, the NIC will see the re-assigned descriptors as a unique batch released by a single thread. As an example: say thread A has granted descriptor 1 and then thread B has granted descriptor 2, but thread A is being slowed down for some reason while B has ended its work. In this case, one can't just write 2 to the TAIL register as this would also mean freeing descriptor 1, which is not done yet. At this point, thread B should wait for thread A for an unknown amount of time, and it has already been stated in Section 6.2 that no waiting periods are allowed. So the only thing B can do to exit the Rx function and process the packet it has received in the meantime is to write to some shared data structure that descriptor 2 can be freed, otherwise, this information would be lost. When thread A eventually ends its routine, it will first write to the same data structure that descriptor 1 is done. Then if there is a contiguous set of descriptors (starting from the current tail onwards) that can be re-assigned to the NIC, it will understand this from reading this shared data structure and will eventually move the TAIL. There could still be multiple threads trying to read this shared data structure and concurrently trying to update the TAIL. Still, the idea is that a thread that sees a continuous batch of descriptors (either by "filling a gap" as thread A did in the previous example or by creating a brand new

one) can give them back to the NIC by updating the TAIL.

6.5 Implementation

In the deployment of this approach, we have found many practical situations that must be taken into account; these are presented here:

1. **Global transaction ID:** A unique ID is needed that tells where the process of assigning descriptors to cores has arrived and that can be updated through the CAS operation at line 21 in Listing 6.1; how to choose it?

   Unfortunately, the naive choice of using the Rx queue descriptor index `queue -> rx_index` at line 8 cannot be done. The reason behind this is that the ID, since it ranges from 0 to `RING_SIZE-1`, is susceptible to the ABA problem[2]; thread A may read index 1023, be descheduled and after some other thread has done a complete round of processing the queue (so the ID is now 1023 again), thread A may wake up again and successfully do a CAS operation, even if it saw an ancient state of the queue! The only solution is, therefore, to use a constantly increasing ID in order to make impossible this periodic wrapping of the index (e.g., using an unsigned 32-bit integer). The assumption here is that the queue size is always a power of 2 to map the ID to the queue positions correctly, but this already happens in network drivers, so it is not limiting the possible Rx queue size in any way. When overflow occurs, the variable will start again from 0, and this does not cause any inconvenience. For mapping the ID to the descriptor offset in the queue, it just need to be divided by the queue

[2]the ABA problem occurs when one thread reads the value A from a shared memory location, some other thread modifies the value from A to B and back to A, and then the first thread observes the value as still being A. The problem is named after the sequence of operations involved (A→B→A). This can lead to unexpected results and incorrect behavior if the first thread misses the intermediate state change.

Listing 6.1: A simplified receive function of a parallel network driver

```
1  #define wrap_ring(index) (uint16_t) (index % RING_SIZE)
2  uint16_t lock = 0;
3
4  uint32_t ixgbe_rx_batch(struct device* dev, uint16_t queue_id, struct
       pkt_buf* bufs[]) {
5    //Get the queue struct for device dev and queue queue_id
6    struct ixgbe_rx_queue* queue = get_queue(dev, queue_id);
7    struct pkt_buf* buffer = queue->buffer;
8    uint16_t rx_index = __atomic_load(queue->rx_index); // descriptor index
         we checked in the last run of this function
9    //Local copy of the rx_index counter
10   uint16_t rx_index_local = rx_index;
11   uint16_t last_rx_index, i;
12   for (i = 0; i < BATCH_SIZE; i++) {
13     if (!(buffer[rx_index_local] && DD_BIT))
14     //Descriptor has not been filled by the NIC yet, exit the loop
15       break;
16     else
17       //Move on to the next descriptor
18       rx_index_local = wrap_ring(rx_index_local + 1);
19   }
20   //try to win the race for the batch of descriptors [rx_index...
         rx_index_local]
21   if (__compare_and_swap(&queue->rx_index, rx_index, rx_index_local)) {
22     //race is won, we can copy the descriptors
23     rx_index_local = rx_index;
24     new_bufs = mempool_desc_bulk_alloc();
25     for (uint16_t j = 0; j < i; j++) {
26       bufs[j] = buffer[rx_index_local];
27       //Replace the descriptor with a new one from the mempool
28       buffer[rx_index_local] = new_bufs[j];
29       last_rx_index = rx_index_local;
30       rx_index_local = wrap_ring(rx_index_local+1);
31     }
32     //write that the [rx_index... rx_index_local] is successfully copied
33     write_batch_is_done(rx_index, rx_index_local);
34   }
35   if (trylock(&lock)) {
36     //Get how many contiguous descriptors there are to be freed, starting
           from the TAIL
37     uint16_t descs_to_free = read_batch_done(queue->tail);
38     //Set the descriptors' bits back to 0
39     write_batch_to_zero(queue->tail, wrap_ring(queue->tail +
         descs_to_free))
40     //Free the processed descriptors back to the NIC
41     set_register(TAIL, dev, queue_id, wrap_ring(queue->tail +
         descs_to_free));
42     release_lock(&lock);
43   }
44 }
```

size and get the rest of the division. The result of the division tells another piece of information, namely the *epoch* in which the queue is (See Table 6.1). The epoch means how many times the system has done a complete round in processing that queue, so from 0 to SIZE-1. On the conceptual level, choosing an ever-growing ID allows one to distinguish between the different epochs the queue may be into, avoiding the previous problem.

2. **How to store in a shared data structure which descriptor has been processed (by any thread) and is ready to be assigned to the NIC?:** It was chosen to use a bitmask with one bit per descriptor called `READ_DONE`: this permits to do the following thing: when all descriptors belonging to a certain iteration have been processed, the thread knows which bits it has to write, and this likely translates into an atomic write to a single variable: in line 33 at Listing 6.1, the thread writes the batch starting from `rx_index` to `rx_index_local`. Bits need to be set not only to 1 at the end of the processing but also to be set back to 0 when a thread grants responsibility for freeing certain descriptors to the NIC (line 40); otherwise, this would cause conflicting views.

6.5.1 Corner cases

The main corner case that may cause the system to stall is preventing the NIC from loading incoming packets to the shared buffer, i.e., the buffer is full. More specifically, say thread A has granted possession of descriptor 2 and gets descheduled for an indefinite period of time. In this situation, other threads can process a full round of descriptors (from 3 to 1). Still, then they will always find the queue full of new descriptors ready to be re-assigned to the NIC but unable to be returned to the

ID	Descriptor Index	Epoch
0	0	
1	1	
2	2	0
...	...	
1023	1023	
1024	0	
1025	1	
1026	2	1
...	...	
2047	1023	
2048	0	
2049	1	
2050	2	2
...	...	
3071	1023	

Table 6.1: Table with the global transaction ID (left), the referred descriptor index (centre) and the consequent epoch (right)

NIC because they lack descriptor 2 in order to form a contiguous batch of descriptors starting from the TAIL. In the meantime, the NIC sees the buffer as full since no thread has had the possibility to move the TAIL. Thus, threads will have to wait for A to resume its execution and mark descriptor 2 as ready to be assigned back to the NIC in the `READ_DONE` shared variable. This is not a limitation strictly caused by the proposed approach but instead by the way in which the NIC-to-CPU communication is designed and explained in Section 2.1.1. We underline that, with respect to the opposed scale-out policy, this approach still permits to perform a full round of operations on the shared buffer, while in the state-of-the-art scale-out policy if one thread is delayed then the whole receive queue(s) assigned to it cannot be processed in any way.

6.5.2 Practical details

The new network driver routine has been implemented in DPDK v21.11. The contribution is not restricted to DPDK but could be extended to other frameworks like the Linux kernel, XDP, or RDMA completion queues. We chose DPDK since it allowed us to write C code in user space and, therefore, ease in deploying the code and debugging it.

We focused on the `ixgbe`, `i40e` and `ice` Intel drivers. The choice of such drivers is motivated by the fact that the `ixgbe` driver is well-documented [27] and well-explained by Emmerich et al. in [19] through their simplified `ixy` driver; `i40e` and `ice` are also pretty similar to `ixgbe` in terms of how the receive function works. Other vendors tend not to show their drivers' routines and specifications through datasheets publicly; we encourage them to make these pieces of information public to increase researchers' interest and knowledge in network drivers.

DPDK drivers usually exploit vectorized ASM instructions in the RX routine in order to optimize performance further by performing the same operation on multiple packets in parallel. These instructions are quite complex to understand [3] firstly because they exploit specific CPU architecture-dependent data structures and APIs. Furthermore, these function routines are not documented by the DPDK developers nor in [19]; in fact, Emmerich et al. limit their contribution to scalar receive functions. For this reason, the vectorized receive function versions were disabled in order to focus only on the standard ones. Still, using vectorized instructions could be of interest since it may show how performances evolve when a better-optimized routine is used.

Regarding the actual code writing, synchronization of the threads is achieved through the `_atomic` [60] and `_sync` [61] `gcc` built-in primitives. `_atomic` functions avoid re-

[3] see the `ixgbe _recv_raw_pkts_vec` function as an example

ordering from out-of-order execution, while the `_sync_bool_compare_and_swap` function performs as a test-and-set primitive: it allows to atomically control if a specific memory location matches a particular value and if the two match, to update the location to a new value.

Chapter 7

Experimental results

In this Chapter, the author presents the evaluation tests for our multithreaded driver. Section 7.1 focuses on how multiple threads scale when bound to the same queue, Section 7.2 shows the results for mean and 99p latency while Section 7.3 quantifies the packet reordering percentage for different traffic sizes. Tests are executed on a server equipped with Intel Silver Xeon 4110 CPU cores clocked at 2.1GHz and Intel XL710 40Gbps NICs. CPU cores are isolated, and power limitators like C-states, P-states are disabled. The sender uses traffic generators like MoonGen [38] or Trex, depending on the test scenario. MoonGen fails to saturate 40Gbps NICs with 64B packets, but it still provides essential features like hardware rate control with Intel X520 NICs (which is not available with Intel XL710) in order to properly quantify the reordering rate. This is why, depending on the scenario, we vary both the traffic generator and the NIC used.

The tested applications are examples included the DPDK framework, namely:

- `l3fwd` [37], which acts as a Layer-3 longest-prefix-matching forwarder. The application retrieves packets from the NIC in batch, executes a routing table lookup for each packet, and forwards it through the relevant NIC.
- `ipsec-gw` gateway [46], which is a more expensive task since it performs more

complex operations; after retrieving a batch of packets, it controls for every packet which rules to apply based on its flow (forward, drop, encryption/decryption), performs the operations and sends the packet if required. In our setup, all packets are encrypted in software without NIC HW acceleration and then forwarded.

7.1 Scalability tests

A first metric worth being measured is how our driver scales in throughput when we add more threads to the same queue. We show the results in Table 7.1 for `l3fwd` and 7.2 for `ipsec-gw`; SOTA stands for the State Of The Art while MT stands for Multithreaded Driver. We show the throughput in Mpps for 64B UDP traffic when executing the tasks on a NIC-local NUMA node and a remote one, as well as the performance improvement in percentage compared to the state of the art. In the case of Table 7.1, the NIC maximum throughput is around 37 Mpps as stated in the Intel XL710 datasheet [44], so a hardware limitation biases this value.

mode	same NUMA		different NUMA	
	Tput (Mpps)	%	Tput (Mpps)	%
SOTA	16.43	100	11.54	100
MT 1 core	17.68	107.61	11.87	102.86
MT 2 cores	26.4	160.68	19.62	170.02
MT 3 cores	35.35	215.16	28.09	243.41
MT 4 cores	37.66	229.21	33.69	291.94

Table 7.1: Scalability executing L3FWD task

A first observation is that our algorithm, also in the 1:1 thread-to-queue comparison, provides some benefits in throughput: the reason behind this is the use of a bulk allocation mechanism from the memory pool (Line 24 in Listing 6.1) which permits to call the allocation only once per Rx routine invocation. A second observation is that

mode	same NUMA		different NUMA	
	Tput (Mpps)	%	Tput (Mpps)	%
SOTA	5.04	100	2.74	100
MT 1 core	5.07	100.6	2.81	102.55
MT 2 cores	8.18	162.3	4.94	180.29
MT 3 cores	10.9	216.27	6.92	252.55
MT 4 cores	15.3	303.57	8.92	325.52

Table 7.2: Scalability executing IPSec task

our algorithm shows better scalability improvements in the remote NUMA scenario; this is a direct consequence of the increased memory access time latencies.

7.2 Latency

We now focus on presenting the end-to-end latency benefits of our approach, motivated by the simulation results in Section 6.1. Our approach (Scale-up) and the state of the art (Scale-out) are compared with different numbers of cores running the `l3fwd` task. We use the Trex traffic generator with minimum-size 64B UDP traffic, as this is the most challenging scenario for DPDK applications in terms of incoming packets rate.

7.2.1 Mean Latency

Figure 7.1 and 7.2 show the mean latency while executing the `l3fwd` task with 4 and 8 cores, respectively, and a variable load, from 0 to the maximum rate sustainable (37Mpps). We can see a similarity between the theoretical plots and the experimental results, as our approach maintains a flat mean latency until the system reaches a saturation point at 37Mpps. Also, we can see that the current scale-out mechanism maintains a flat mean latency until a certain threshold is reached. After that threshold, the value rapidly spikes to a saturation value. As from the simulation results, we would have expected the mean latency to increase up to saturation slightly, this

likely does not happen both because of the highly optimized DPDK threads and also because of the little service time needed to perform level-3 forwarding. Instead, our scale-up approach has the benefit of bringing further the saturation point up to the theoretical maximum achievable from the system, as also queuing theory and simulation suggest. Indeed, in this case, the latency spike at 37Mpps is not caused by software but rather by the NIC hardware limitations, so with different hardware, the saturation point could possibly be pushed forward (especially in the 4 cores case) and show more significant improvements.

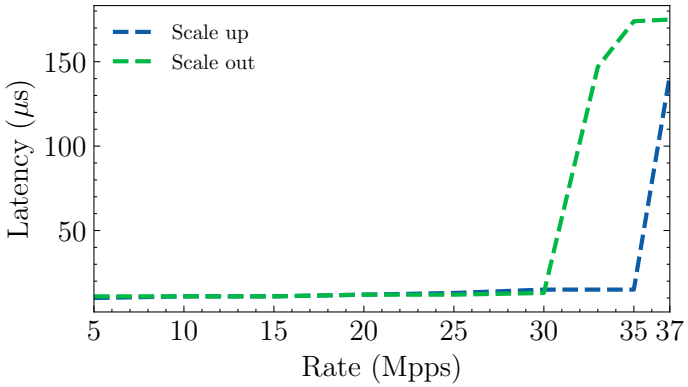


Figure 7.1: L3FWD mean latency - 4 cores

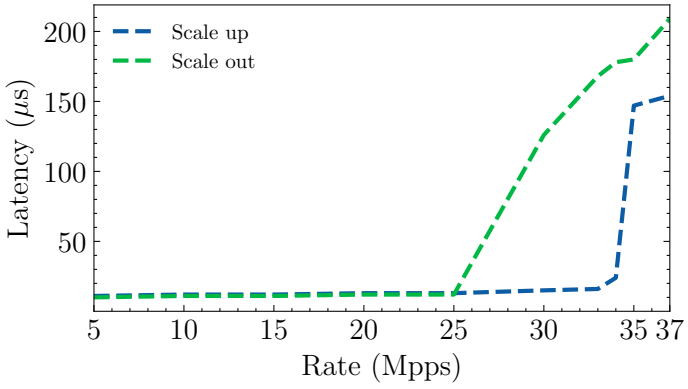


Figure 7.2: L3FWD mean latency - 8 cores

7.2.2 Tail Latency

We now focus on rates where our scale-up policy achieves better mean performance than the classical scale-out mechanism[1]. Figure 7.3 shows the CDF latency distributions at 35Mpps with 4 cores, while Figure 7.4 shows the same lines with 8 cores and a 30Mpps traffic. The two figures clearly show that our approach brings benefits not only in terms of mean latency but, most of all, latency predictability.

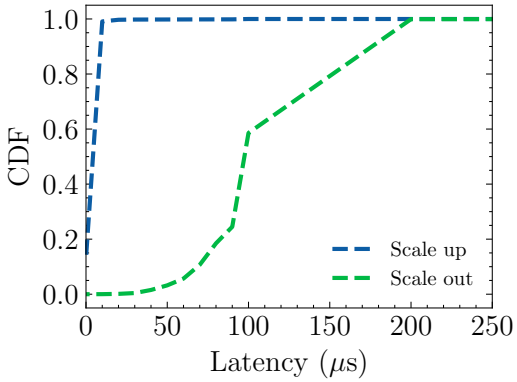


Figure 7.3: L3FWD latencies CDF distribution - 35Mpps

7.3 UDP reordering

These tests focus on how the multithreaded driver impacts connection-less streams like UDP ones, as they are a good way of showing a worst-case scenario for the reordering metrics. Tests for reordering with UDP traffic are performed with different-sizes packets on a 10Gb link. The test focuses on sending a unique flow of 100k

[1]It is possible that further improvements in tail latency may be shown: the reason behind this is the limited resolution of the TRex traffic generator [62]. At the moment, TRex is the only framework capable of saturating a 40Gbps NIC while permitting end-to-end latency measurements at the same time. Unfortunately, TRex does not use the same precise NIC hardware timestamp features as MoonGen [38] does. Moreover, the latency distribution is shown in a histogram mode with bins, so there is no way to get the exact latency values for a precise histogram unless the TRex code is modified. The author had a conversation with the authors of TRex about this [63] and believes this limitation could be overcome either by modifying TRex or by building a new traffic generator from scratch.

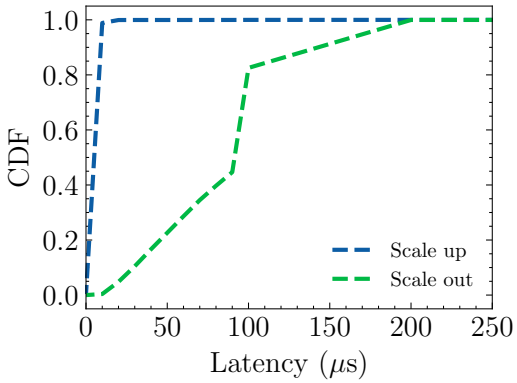


Figure 7.4: L3FWD latencies CDF distribution - 30Mpps

sequenced packets, making them reordered by the MT driver through a L3 forwarder, and checking the order of the arrived packets at the receiver side. Since it is a unique flow, we do not include results for the scale-out policy, as the reordering percentage would be zero regardless of the rate or packet size. Results are shown in Figure 7.5 with 4 cores pinned to the same Rx queue and in Figure 7.6 with 8 cores. The plots focus on the percentage of packets reordered, according to RFC 4737 [64]. It can be clearly seen that high levels of packet reordering are achieved only in the presence of both high traffic rates and minimal packet sizes. In fact, as the packet sizes increase, the reordering percentage rapidly drops and becomes insignificant for typical packet sizes like TCP ones.

7.4 TCP

We now investigate how a `l3fwd` router equipped with our multithreaded driver can impact real-world use cases using TCP connections to exchange data. More specifically, the router is connected to a client and a server and forwards traffic between the two. The router is deployed in two different scenarios, scale-out (like the state of the art) and scale-up (our solution). Tests are executed with 1, 2 or 4 threads per NIC:

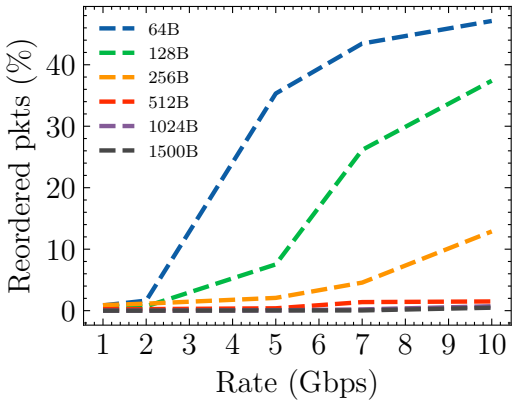


Figure 7.5: percentage of reordered packets for UDP traffic of different sizes with 4 cores

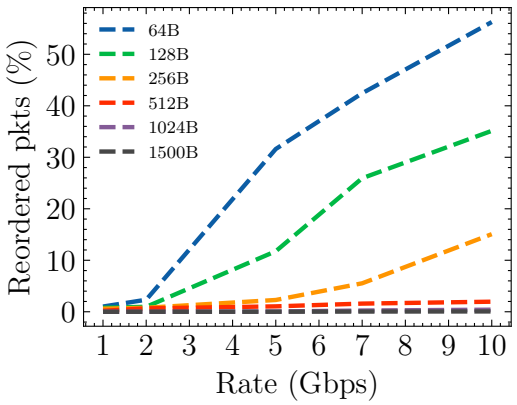


Figure 7.6: percentage of reordered packets for UDP traffic of different sizes with 8 cores

as explained, in the scale-out option each thread has its own Rx queue, while in the scale-up option one queue is shared among all the threads. Our goal is to investigate, in different test scenarios, whether our approach can improve the end-to-end latency or can cause performance degradation because of the retransmissions caused by re-ordering. We used the standard Linux TCP CUBIC congestion control.

We focus on three different test scenarios: one massive flow (High-Performance-Computing-style), many medium flows (ordinary client connections), and many small flows (data-center RPC-style connections). The first case is not compared to the scale-

mode	**FCT (s)**		**Retransmissions**	
	1GB	**10GB**	**1GB**	**10GB**
COREC 1 core	0.91625	9.13267	6.375	906.25
COREC 2 cores	0.91853	9.2238	626	4073.88
COREC 4 cores	0.92067	9.35045	1071.25	5042

Table 7.3: Latency and # retransmissions for huge flows

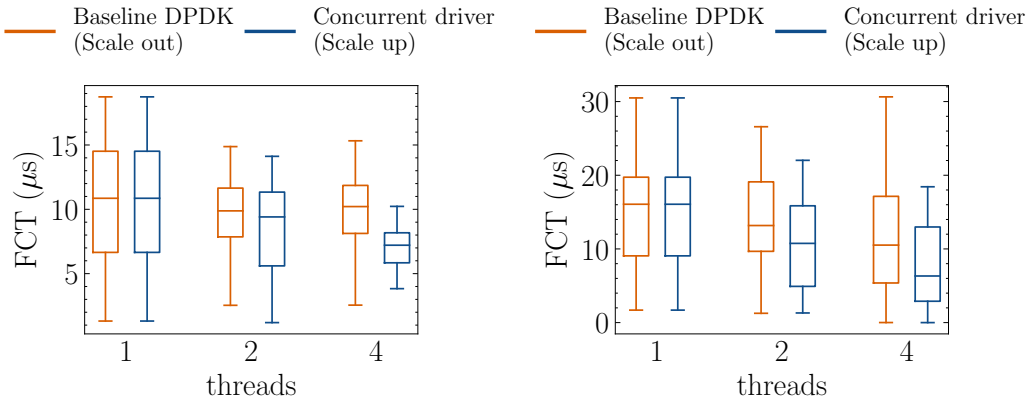
out policy since a unique flow would involve only one Rx queue and thread because of RSS; therefore, there is no chance to distribute the load among cores.
Latencies are calculated by retrieving the OS timestamp right before the connection setup (`connect` syscall) and right after the connection teardown (`close` syscall).

Single Huge Flow: We test our approach with two different flow sizes, namely 1GB and 10GB. This is expected to be the worst case for our COREC driver since any packet reordering occurs within the single flow being delivered, and thus directly impacts the TCP transmission control protocol. Results are shown in Table 7.3. As expected, our approach causes performance degradation in the Flow Completion Time (FCT), owing to the increase in the TCP retransmissions, which are a direct consequence of packet reordering and are also exacerbated when the whole bandwidth of the 10Gbps link is assigned to a unique flow. Still, performance degradation is marginal, with an increase in the flow completion time of 2.3% in the case of 10GB flow when moving from 1 thread to 4 threads, and even less than 1% in the 1GB experiment.

Medium and small flows: Tests are executed with a 100 KB and 10KB payload per connection. We first comment on the results for the 10KB connections shown in Figure 7.8a for 64 flows and Figure 7.8b for 128 flows. We can clearly see that our approach brings significant benefits both in mean and tail latency by exploiting its work-conserving capabilities. On the other side, the scale-out case does not scale so

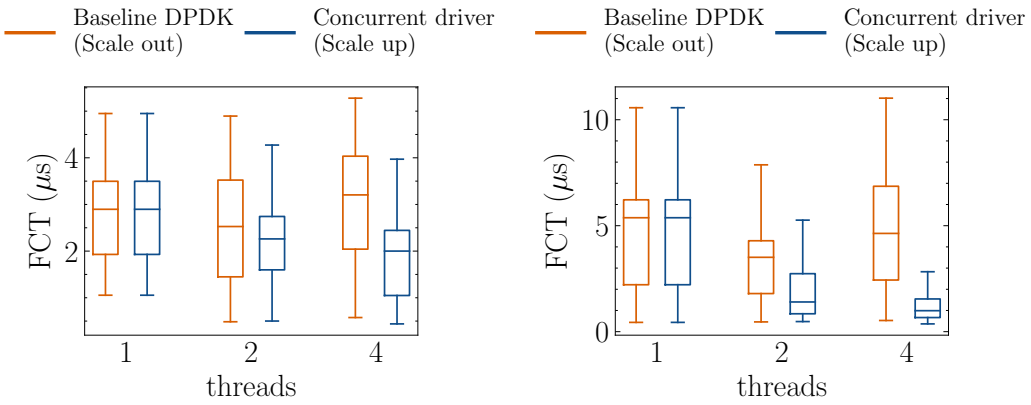
well to multiple cores, confirming the theory of poor many-core scaling described in [17].

While in this case we can expect that reordering is very unlikely because of the short payload, we now try to increase the payload to 100KB, as this case focuses on a more significant load of ∼70 packets per flow. Results are presented in Figures 7.7a and 7.7b, and they clearly show that our driver still improves the FCT, although in a less evident way.



(a) 64 flows (b) 128 flows

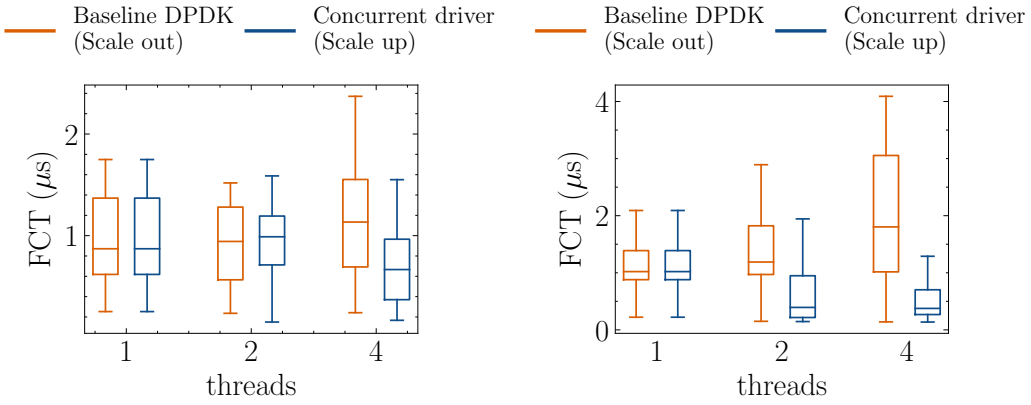
Figure 7.7: Flow Completion Time with 100KB payload



(a) 64 flows (b) 128 flows

Figure 7.8: Flow Completion Time with 10KB payload

One-packet flows: It is interesting to also test our driver with one-packet flows (1KB payload). This can be considered as a best case benchmark for TCP traffic, since, in this case, there is no possibility of re-sequencing the packets at the receiver since only one packet containing TCP payload is sent. Furthermore, these flows are particularly interesting since a significant portion of Data Center flows, especially RPC flows, are restricted to a single packet [54, 65]. Also, in this case, we test the FCT time for 64 and 128 TCP parallel flows, and results are shown in Figures 7.9a and 7.9b. We can clearly see the benefits of our multithreaded driver as the number of flows increases.



(a) 64 flows

(b) 128 flows

Figure 7.9: Flow Completion Time with 1KB payload

Part III

Related Work

Chapter 8

Related Work

8.1 Optimized networking software

8.1.1 Latency

In the last years, a set of works were published with the common goal of such solutions is the efficient schedule of tasks across different cores, explicitly targeting interference mitigation and predictable, low latency for user applications. These works focus on different components in order to increase performances:

- **a CPU scheduler**: Shenango [66] permits dynamic fast core reallocation between latency-sensitive applications and batch ones by detecting spikes in queuing delay, where the latter can exploit the unused CPU cycles of the former. Caladan [67] is a similar work with respect to Shenango but uses more signals for detecting interference and uses a dedicated kernel module for fast reallocation. The work in [68] builds on Caladan to investigate different reallocation policies and shows improvements in CPU efficiency.

- **user-level thread-management system:** Arachne [69] is a core-aware user-level thread implementation that minimizes cache misses and achieves low-latency high-throughput performance for short-lived threads. The paper in [70]

presents a user-level M:N threading runtime that enables thread-per-session programming for long-running communication applications. The runtime features efficient load balancing and user-level I/O blocking.

- **a performance-isolation framework:** [71] PerfIso is a performance isolation framework used by the Microsoft Bing search engine to colocate batch jobs with production latency-sensitive services on over 90,000 servers. By doing so, idle resources can be used to run batch jobs without compromising the stringent tail-latency requirements of the latency-sensitive services. The experimental evaluation conducted in a production environment shows that PerfIso increases average CPU utilization without impacting tail latency.;

- **a specific OS:** ZygOS [72] is an operative system optimized for high-throughput, low-latency scheduling of fine-grain networked tasks on multicore servers. ZygOS achieves high performance for tail latency service-level objectives, using a work-conserving scheduler, shared-memory data structures, multi-queue NICs, and inter-processor interrupts to rebalance work across cores. Shinjuku [73] is an operating system with fast preemption at the microsecond scale enabled by hardware virtualization, enabling centralized scheduling policies for short service time requests. IX [74] is designed for high I/O performance with strong protection. It uses hardware virtualization to separate control and dataplane, with a zero-copy API, dedicated hardware threads, and networking queues to optimize for bandwidth and latency.

All of the presented works try to minimize tail latency by focusing on the application level; instead, this dissertation dives deep into one of the datapath components, namely the network driver. Moreover, my proposal may cause, as an advantage, a

better use of the CPU caches because of the reduced memory footprint of one queue compared to many queues, and could be further coupled with one of the approaches in Section 8.1.2.

8.1.2 Cache performances

As networking moves towards the Terabit Ethernet, a smart and timely use of cache hierarchies becomes fundamental [75, 76]. Direct cache Access (DCA) permits the NIC to place DMAed buffers directly in the L3 cache so that software can find a warm cache, with Intel's DDIO [77] being the most popular solution [78]. However, several works showed that DDIO is not a panacea at all: Cai et al. [79] *"observe that it suffers from high cache miss rates (49%) even for a single flow"*, while [80, 81] present optimized DDIO versions in terms of latency. In the context of more NFs sharing the same L3 cache, [82] proposes a limitation on the number of descriptors for avoiding the *leaky DMA* problem, where LLC cache contention can cause the eviction of packets which still have to be processed by the system. The work in [83] shows how delaying and reordering packets can improve cache locality and, therefore, performance.

8.1.3 Power consumption

One of the main shortcomings of DPDK is the excessive usage of resources (CPU cycles and energy), caused by the busy-wait approach used by threads to check the state of NICs and Rx queues. Intel tried in [84] to address the energy consumption issue via a gradual decrease of the CPU clock frequency under low traffic for a commonly used

application such as the layer-3 forwarder. A similar approach is used in [85], with the addition of an analytical model exploited to choose the appropriate CPU frequency. Along this line, [4] proposes a power-proportional software-router.

However, while the downgrading of the clock frequency reduces power consumption [4] without noticeably affecting performance, these solutions do not take into account another crucial aspect, namely the actual usage of CPU. In fact, downgrading the clock frequency of a CPU core fully dedicated to a thread operating in busy-wait (namely, continuous polling) mode still implies 100% utilization. Hence, the CPU core is anyhow unusable for other tasks. Moreover, downgrading the clock frequency of CPUs is not feasible in cloud environments since (i) they are shared between different processes and customers and (ii) providers would like them to be fully utilized in order to reach peak capacity on their servers [1]. On the contrary, the author's proposal, Metronome, bypasses these limitations since it does not rely on any explicit manipulation of the frequency and/or power state of the CPUs. Instead, it controls at fine-grained the timeline of CPU (and energy) usage by DPDK threads—hence the name Metronome—which are no longer required to operate in busy-wait style. Such control is based on an analytical model that allows taking runtime decisions depending on packet workload variations.

8.1.4 Other optimizations

Recent works propose low-level code optimizations, either at run time [86] or through a tailored binary file [87]. It is essential to underline that, while the NIC and the CPU exchange packets, the PCI can possibly become a bottleneck in the communication process [88].

8.2 Network drivers

It has been quite hard for the author to understand what happens under the hood of a network driver. The reason behind this is that these components are usually seen as black boxes by researchers, mainly because:

- they are usually developed in the industry with poor documentation [55];
- modern frameworks abstract the details to the programmer for simplicity reasons, thus simply exposing a function to receive/transmit packets.

However, in recent years, some works have tried to explain and improve what happens under the hood of a network driver. `ixy` [19] is a simple implementation of the `ixgbe` network driver, with simplicity and educational goals, and has been widely studied by the author to get a primary idea of how drivers work. TinyNF [55] simplifies the packet handling for the same driver, showing increased performances and more simplicity at the cost of some flexibility. The work by Emmerich et al. [56] proposes writing network drivers in higher-level languages than standard C/C++ implementations to ensure memory safety, reduce bugs and, more broadly, exploit the unique features of each different language. CleanQ [89] is a reusable formalization of a driver's descriptor ring with security and portability motivations.

8.2.1 Modern optimizations

DMA (Direct Memory Access) is a technique used by network drivers to improve data transfer between the NIC and the operating system. DMA allows the NIC to access the system's memory directly, without the need for intervention by the CPU. This technique can reduce the CPU overhead and improve performance by allowing the NIC to move data directly to or from memory. In essence, DMA allows the NIC

to act as a co-processor, handling data transfers independently of the CPU.

IOMMU (Input/Output Memory Management Unit) is another technique used by network drivers to improve data transfer between the NIC and the operating system. IOMMU is a hardware component that allows the system to map virtual addresses to physical addresses. This enables the NIC to access memory without the risk of accessing data outside its allocated memory space. This technique is particularly useful in virtualized environments where multiple virtual machines are running on a single physical server.

8.3 Non-blocking algorithms

As a notable example, Read-Copy-Update (RCU) [90] is used in *many readers, few writers* situations and is widely adopted also in the Linux Kernel [91]. Read-Log-Update [49] is a novel version of RCU designed with the goal of also enabling concurrent writes. The work by Harris [53] proposes a non-blocking implementation of a linked list by exploiting the compare-and-swap primitive. NBBS [50] is a novel implementation of a buddy system based on read-modify-write (RMW) primitives. Anonymous Readers Counting (ARC) [57] exploits RMW instructions to enable a vast number of concurrent readers in multi-word atomic registers. [58] exploits these non-blocking approaches to design a Parallel Discrete Event Simulation system. Similar approaches have also been used to increase performances in binary search trees, either through lock-free RMW instructions [51, 52] or transactional memory [59].

Part IV

Concluding Remarks

Chapter 9

Conclusions

In this Dissertation, we explored the new challenges and requirements brought out by the recent advantage in NIC speeds, with a particular focus on data centers. Indeed, the stagnation in CPU performance improvements has caused a shift towards specifically-designed network stacks and frameworks, like DPDK. The deployment of such frameworks is particularly needed for typical data center applications, which call for high throughput and low latency. Also, the sustainability theme regarding data center consumption is a tedious problem and will become more important in the upcoming years, and it indeed requires a tradeoff between power and performance.
This Dissertation has proposed the design and implementation of two different strategies in order to address the aforementioned challenges:

- Metronome for CPU-proportional, cloud-resilient network frameworks;
- A non-blocking, parallel network driver to meet the demands for scalability and low latency.

In the first strategy, Metronome has the goal of replacing the continuous and CPU-consuming DPDK polling with a sleep&wake, load-adaptive, intermittent packet retrieval mode. Metronome's viability has been evaluated by integrating it into three

different typical DPDK applications and by showing its significant improvements primarily in terms of CPU utilization (and, partially, also in terms of power consumption), and therefore its ability to release precious CPU cycles to business applications. In the future, deploying Metronome in multi-NICs scenarios with per-NUMA node thread pools could be considered as a further step. The author also stresses that such gains are traded off with an extra latency toll, which can be taken into account and configured using the tuning knobs provided by this approach, especially when (and if) considering the usage of Metronome with time-critical applications.

In the second strategy, the design of a non-blocking, parallel network driver has been motivated both by the native beneficial properties of scale-up systems and by the curiosity about applying techniques from other contexts in the networking environment. The challenges presented in the designed algorithm required not only low-overhead threads coordination but also transparency and compatibility with existing NICs. The adoption of the non-blocking, parallel network driver has shown significant improvements in terms of mean latency and tail latency. These advantages could be even more considerable whenever this approach is also tested on hundred-gigabit NICs, where more considerable degrees of scalability are expected.

In the future, there may be a possibility for a fusion of the Metronome architecture with the non-blocking driver. Indeed, this contribution would not be trivial as it would require a deep re-architecture of the Metronome adaptive model, posing the critical question of how the changes in the queuing model impact Metronome's adaptive capabilities.

Also, these contributions could be ported to different contexts, like the Linux Kernel or RDMA. As better explained in Section 1.3.1, this porting may not be straightforward and therefore requires a deep engineering effort. Moreover, the non-blocking

driver could be implemented in more NIC models, like the Mellanox/NVIDIA ones, whether these vendors will release their datasheets publicly along with comprehensive network drivers documentation.

The author hopes that the combination of the works presented in this Dissertation can make a contribution in paving the way for a new generation of frameworks and algorithms, which should provide most of the characteristics proposed in this Dissertation, like scalability, CPU-proportionality, power-saving, resource sharing, low-level coordination, dynamic scaling, and predictable latency.

Acknowledgements

This Dissertation is the result of years of hard work, dedication, and passion in the SDN-NFV Laboratory in Tor Vergata. This work has been made possible through the incredible effort, vision, and guidance of my supervisors, Prof. Giuseppe Bianchi and Prof. Francesco Quaglia. Among the many things I learned from them, I can say that they taught me to have a critical view of research, focusing not on the achievements (publications, awards) but rather on the techniques and approaches used. In other words, research is a journey without knowing the destination, so one should simply enjoy the journey, enriching it with new ideas and experiences.

Giuseppe is the man who first introduced me to the research world and gave me the chance to prove my skills; I will always be grateful to him for giving me this opportunity. He inspires me every day and pushes me to overcome my limits, raising the bar each time. I firmly believe Giuseppe is the best role model that a new generation of brilliant, young researchers could ask for.

Francesco managed to push me each time I had doubts and struggled to find meaning in my work. His help was fundamental in finding critical aspects of the approaches followed in the networking community, and proposing meaningful solutions coming from his background in parallel and high-performance computing.

A fundamental contribution to my career and also to this Dissertation has been made through the help of Prof. Salvatore Pontarelli. Sharing the lab with him was a true

privilege, and his everyday supervision has been extremely helpful in my growth.
Giacomo Belocchi was my main colleague for the past three years; our day-to-day activity went far beyond what I could expect, from bug-solving to brainstorming about possible research directions. Most of all, he is a true friend who was able to share with me his sincere passions and doubts. Giacomo is the smartest and most passionate guy I've ever met in this field, and I'm sure he has a bright future ahead of him.
I also want to thank my other colleagues and friends from the Topini group: Daniela, Marco, Alessandro, Angelo, Aniello, Emanuele, Luca, Ludovico, Marco, Valerio: I always went to lab knowing that it was a place full of joy and that is because of these lovely people.
My three months at MSR Cambridge were far beyond what I could imagine from an internship. A huge "thank you" goes to Sergey Legtchenko, Paolo Costa, the whole Silica team, and all the MSR Residents. I had the chance not only to work on amazing, real-life projects but, most of all, to meet lovely and caring people from all over the world, and I will always keep this experience in my heart.

Proseguirò ora in italiano per i ringraziamenti personali.
Grazie ad Alessandra, persona speciale ed anima buona, che più di tutti mi è stata accanto. Qualsiasi parola non renderebbe bene cosa siamo.
Grazie ai miei genitori Cristina e Fabio, ed a mio fratello Alessandro. Credo fermamente che la famiglia ci influenzi profondamente nella nostra crescita: i valori, l'educazione e tutti quei sistemi complessi che regolano il nostro stare al mondo. Voi siete il meglio che la vita potesse darmi.
Grazie ai miei amici Burini, famiglia e porto sicuro dove tornare sempre.
Grazie a mio nonno Enzo, ingegnere da una vita e che fin da piccolo ha stimolato la

mia curiosità, insegnandomi a porre le giuste domande. È proprio vero che la mela non cade lontana dall'albero.

Auguro a chiunque legga questa tesi di sentirsi vivo mentre segue le proprie passioni, così come è capitato a me in questo percorso.

Beauty in the struggle, ugliness in the success

Bibliography

[1] Daniel Firestone et al. "Azure accelerated networking: SmartNICs in the public cloud". In: *USENIX NSDI 2018*.

[2] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. "Large-scale cluster management at Google with Borg". In: *Proceedings of the Tenth European Conference on Computer Systems*. 2015, pp. 1–17.

[3] John L Hennessy and David A Patterson. "A new golden age for computer architecture". In: *Commun. of the ACM* 62.2 (2019), pp. 48–60.

[4] Luca Niccolini, Gianluca Iannaccone, Sylvia Ratnasamy, Jaideep Chandrashekar, and Luigi Rizzo. "Building a power-proportional software router". In: *USENIX ATC 2012*.

[5] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. "The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition". In: 2018. Chap. 1.6.5.

[6] *NVIDIA BlueField-3 Datasheet*. URL: `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf` (visited on 11/17/2021).

[7] *Ethernet Roadmap 2022*. URL: `https://ethernetalliance.org/technology/ethernet-roadmap/` (visited on 01/31/2023).

[8] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network". In: *ACM SIGCOMM computer communication review* 45.4 (2015), pp. 183–197.

[9] *DPDK*. Linux Foundation. URL: `https://www.dpdk.org/` (visited on 05/25/2020).

[10] Jeffrey Dean and Luiz André Barroso. "The tail at scale". In: *Communications of the ACM* 56.2 (2013), pp. 74–80.

[11] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. "Tales of the tail: Hardware, os, and application-level sources of tail latency". In: *Proceedings of the ACM Symposium on Cloud Computing*. 2014, pp. 1–14.

[12] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. "ProjecToR: Agile Reconfigurable Data Center Interconnect". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. 2016, pp. 216–229.

[13] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. "Inside the social network's (datacenter) network". In: *Proceedings of the 2015 ACM SIGCOMM Conference*.

[14] Theophilus Benson, Aditya Akella, and David A Maltz. "Network traffic characteristics of data centers in the wild". In: *ACM IMC 2010*.

[15] Laxmana Rao Battula. *DPDK (Data Plane Development Kit) for Linux VMs now generally available*. Microsoft Azure, Sept. 21, 2018. URL: https://azure.microsoft.com/en-us/blog/dpdk-data-plane-development-kit-for-linux-vms-now-generally-available/ (visited on 05/25/2020).

[16] Jeff Barr. *Elastic Network Adapter – High Performance Network Interface for Amazon EC2*. Amazon Web Services, June 28, 2016. URL: https://aws.amazon.com/it/blogs/aws/elastic-network-adapter-high-performance-network-interface-for-amazon-ec2/ (visited on 05/25/2020).

[17] Hossein Golestani, Amirhossein Mirhosseini, and Thomas F Wenisch. "Software data planes: You can't always spin to win". In: *Proceedings of the ACM Symposium on Cloud Computing*. 2019, pp. 337–350.

[18] Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle. "Comparison of frameworks for high-performance packet IO". In: *ACM/IEEE ANCS 2015*.

[19] Paul Emmerich, Maximilian Pudelko, Simon Bauer, Stefan Huber, Thomas Zwickl, and Georg Carle. "User space network drivers". In: *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE. 2019, pp. 1–12.

[20] Marco Faltelli, Giacomo Belocchi, Francesco Quaglia, Salvatore Pontarelli, and Giuseppe Bianchi. "Metronome: Adaptive and Precise Intermittent Packet Retrieval in DPDK". In: *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '20. Barcelona, Spain: ACM, 2020, pp. 406–420. DOI: 10.1145/3386367.3432730.

[21] Marco Faltelli, Giacomo Belocchi, Francesco Quaglia, Salvatore Pontarelli, and Giuseppe Bianchi. "Metronome: Adaptive and Precise Intermittent Packet Retrieval in DPDK". In: *IEEE/ACM Transactions on Networking* (2022), pp. 1–15. DOI: 10.1109/TNET.2022.3208799.

[22] Marco Faltelli, Giacomo Belocchi, Giuseppe Bianchi, and Francesco Quaglia. *A non-blocking, parallel driver for low and predictable latency*. Tech. rep. Consorzio Nazionale Interuniversitario per le Telecomunicazioni - CNIT, 2022.

[23] Valerio Bruschi, Marco Faltelli, Angelo Tulumello, Salvatore Pontarelli, Francesco Quaglia, and Giuseppe Bianchi. "Offloading Online MapReduce tasks with Stateful Programmable Data Planes". In: *IEEE NETPROC 2020*, pp. 17–22. DOI: 10.1109/ICIN48450.2020.9059417.

[24] Giuseppe Bianchi, Marco Faltelli, and Valerio Bruschi. "Back to the Future: Towards Hardware "Netputing" Architectures". In: *IEEE MedComNet 2020*, pp. 1–4. DOI: 10.1109/MedComNet49392.2020.9191475.

[25] Jeffrey C. Mogul and K. K. Ramakrishnan. "Eliminating Receive Livelock in an Interrupt-Driven Kernel". In: *ACM Transactions on Computer Systems* 15.3 (Aug. 1997), pp. 217–252.

[26] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux device drivers*. O'Reilly Media, Inc., 2005. Chap. 17: Networking drivers.

[27] *Intel® 82599 10 GbE Controller Datasheet Rev. 3.4*. Section 7.1.9. Intel, Nov. 2019. URL: https://www.intel.com/content/www/us/en/embedded/products/networking/82599-10-gbe-controller-datasheet.html.

[28] Luigi Rizzo. "netmap: A Novel Framework for Fast Packet I/O". In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA, 2012, pp. 101–112.

[29] N. Bonelli, S. Giordano, and G. Procissi. "Network Traffic Processing With PFQ". In: *IEEE Journal on Selected Areas in Communications* 34.6 (2016), pp. 1819–1833.

[30] *PF_RING ZC (Zero Copy)*. ntop, 2014. URL: https://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/ (visited on 05/13/2020).

[31] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. "The eXpress Data Path: Fast programmable packet processing in the operating system kernel". In: *Proceedings of the 14th international conference on emerging networking experiments and technologies*. 2018, pp. 54–66.

[32] *eBPF*. https://ebpf.io/.

[33] Maurice Herlihy and J. Eliot B. Moss. "Transactional Memory: Architectural Support for Lock-Free Data Structures". In: *SIGARCH Comput. Archit. News* 21.2 (May 1993), pp. 289–300. ISSN: 0163-5964. DOI: 10.1145/173682.165164. URL: https://doi.org/10.1145/173682.165164.

[34] Matt Mills. *Transactional Memory: What It Is and Integration Via Intel TSX*. July 9, 2021. URL: https://itigic.com/transactional-memory-what-it-is-and-integration-via-intel-tsx/.

[35] *Metronome: adaptive packet retrieval in DPDK*. URL: %7Bhttps://github.com/marcofaltelli/Metronome%7D (visited on 10/31/2020).

[36] Muthurajan Jayakumar. *Data Plane Development Kit (DPDK)—Multicores and Control Plane Synchronization*. Intel, Sept. 10, 2018. URL: https://software.intel.com/content/www/us/en/develop/articles/dpdk-data-plane-multicores-and-control-plane-synchronization.html (visited on 06/08/2020).

[37] *Sample Applications User Guides - L3 Forwarding Sample Application*. URL: https://doc.dpdk.org/guides-19.11/sample_app_ug/l3_forward.html (visited on 06/28/2020).

[38] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. "MoonGen: A Scriptable High-Speed Packet Generator". In: *ACM IMC 2015*.

[39] Tianzhu Zhang, Leonardo Linguaglossa, Massimo Gallo, Paolo Giaccone, Luigi Iannone, and James Roberts. "Comparing the Performance of State-of-the-Art Software Switches for NFV". In: *ACM CoNEXT 2019*.

[40] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. "RAPL in Action: Experiences in Using RAPL for Power Measurements". In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* 3.2 (2018).

[41] *CPU frequency and voltage scaling code in the Linux kernel*. URL: https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt (visited on 04/01/2021).

[42] *Achieving line rate with xdp_fwd using Intel X520 #53*. URL: https://github.com/xdp-project/xdp-project/issues/53 (visited on 10/19/2020).

[43] Christian Bienia. "Benchmarking Modern Multiprocessors". PhD thesis. Princeton University, Jan. 2011.

[44] *Intel® Ethernet Controller X710/XXV710/XL710 Specification Update, Section 2, Clarification #13*. URL: https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xl710-10-40-controller-spec-update.pdf (visited on 04/01/2021).

[45] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. "The Design and Operation of CloudLab". In: *2019 USENIX annual technical conference (USENIX ATC 19)*. 2019, pp. 1–14.

[46] *Sample Applications User Guides - IPsec Security Gateway Sample Application*. URL: https://doc.dpdk.org/guides-19.11/sample_app_ug/ipsec_secgw.html (visited on 06/28/2020).

[47] T. Zhang, L. Linguaglossa, M. Gallo, P. Giaccone, and D. Rossi. "FloWatcher-DPDK: Lightweight Line-Rate Flow-Level Monitoring in Software". In: *IEEE Transactions on Network and Service Management* 16.3 (2019), pp. 1143–1156.

[48] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. "mtcp: a highly scalable user-level TCP stack for multicore systems". In: *USENIX NSDI 2014*.

[49] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. "Read-log-update: a lightweight synchronization mechanism for concurrent programming". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. 2015, pp. 168–183.

[50] Romolo Marotta, Mauro Ianni, Alessandro Pellegrini, and Francesco Quaglia. "NBBS: A non-blocking buddy system for multi-core machines". In: *IEEE Transactions on Computers* (2021).

[51] Aravind Natarajan and Neeraj Mittal. "Fast concurrent lock-free binary search trees". In: *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 2014, pp. 317–328.

[52] Bapi Chatterjee, Nhan Nguyen, and Philippas Tsigas. "Efficient lock-free binary search trees". In: *Proceedings of the 2014 ACM symposium on Principles of distributed computing*. 2014, pp. 322–331.

[53] Timothy L Harris. "A pragmatic implementation of non-blocking linked-lists". In: *International Symposium on Distributed Computing*. Springer. 2001, pp. 300–314.

[54] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. "Homa: A receiver-driven low-latency transport protocol using network priorities". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 2018, pp. 221–235.

[55] Solal Pirelli and George Candea. "A Simpler and Faster NIC Driver Model for Network Functions". In: *14th USENIX Symposium on Operating Systems Design and Implementation OSDI 20)*. 2020, pp. 225–241.

[56] Paul Emmerich, Simon Ellmann, Fabian Bonk, Alex Egger, Esaú García Sánchez-Torija, Thomas Günzel, Sebastian di Luzio, Alexandru Obada, Maximilian Stadlmeier, Sebastian Voit, and Georg Carle. "The Case for Writing Network Drivers in High-Level Programming Languages". In: *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 2019, pp. 1–13. DOI: 10.1109/ANCS.2019.8901892.

[57] Mauro Ianni, Alessandro Pellegrini, and Francesco Quaglia. "Anonymous Readers Counting: A Wait-Free Multi-Word Atomic Register Algorithm for Scalable Data Sharing on Multi-Core Machines". In: *IEEE Transactions on Parallel and Distributed Systems* 30.2 (2019), pp. 286–299. DOI: 10.1109/TPDS.2018.2865932.

[58] Mauro Ianni, Romolo Marotta, Davide Cingolani, Alessandro Pellegrini, and Francesco Quaglia. "The Ultimate Share-Everything PDES System". In: *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. SIGSIM-PADS '18. Rome, Italy: Association for Computing Machinery, 2018, pp. 73–84. ISBN: 9781450350921. DOI: 10.1145/3200921.3200931. URL: https://doi.org/10.1145/3200921.3200931.

[59] Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. "A practical concurrent binary search tree". In: *ACM Sigplan Notices* 45.5 (2010), pp. 257–268.

[60] *Built-in Functions for Memory Model Aware Atomic Operations*. GCC, the GNU Compiler Collection. URL: https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html#g_t_005f_005fatomic-Builtins (visited on 08/16/2022).

[61] *Legacy __sync Built-in Functions for Atomic Memory Access*. GCC, the GNU Compiler Collection. URL: https://gcc.gnu.org/onlinedocs/gcc/_005f_005fsync-Builtins.html#g_t_005f_005fsync-Builtins (visited on 08/16/2022).

[62] *TRex*. URL: trex-tgn.cisco.com (visited on 10/12/2022).

[63] *Trex per-packet latency*. URL: https://groups.google.com/g/trex-tgn/c/9ZEvJyx_lOQ/m/iFo8TOuIAwAJ (visited on 10/24/2022).

[64] Al Morton, Gomathi Ramachandran, Stanislav Shalunov, Len Ciavattone, and Jerry Perser. *Packet Reordering Metrics*. RFC 4737. Nov. 2006. DOI: 10.17487/RFC4737. URL: https://www.rfc-editor.org/info/rfc4737.

[65] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. "Dagger: Efficient and Fast RPCs in Cloud Microservices with near-Memory Reconfigurable NICs". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '21. Virtual, USA: Association for Computing Machinery, 2021, pp. 36–51. ISBN: 9781450383172. DOI: 10.1145/3445814.3446696. URL: https://doi.org/10.1145/3445814.3446696.

[66] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. "Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads". In: *USENIX NSDI 2019)*.

[67] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. "Caladan: Mitigating Interference at Microsecond Timescales". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 281–297. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/fried.

[68] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. "Efficient Scheduling Policies for Microsecond-Scale Tasks". In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022, pp. 1–18. ISBN: 978-1-939133-27-4. URL: https://www.usenix.org/conference/nsdi22/presentation/mcclure.

[69] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. "Arachne:{Core-Aware} Thread Management". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 145–160.

[70] Martin Karsten and Saman Barghi. "User-Level Threading: Have Your Cake and Eat It Too". In: *Proc. ACM Meas. Anal. Comput. Syst.* 4.1 (June 2020). DOI: 10.1145/3379483. URL: https://doi.org/10.1145/3379483.

[71] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. "PerfIso: Performance Isolation for Commercial Latency-Sensitive Services". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 519–532. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/iorgulescu.

[72] George Prekas, Marios Kogias, and Edouard Bugnion. "ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks". In: *ACM SOSP 2017*.

[73] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. "Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency". In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 2019, pp. 345–360.

[74] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. "The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane". In: *ACM Trans. Comput. Syst.* 34.4 (Dec. 2016). ISSN: 0734-2071. DOI: 10.1145/2997641. URL: https://doi.org/10.1145/2997641.

[75] Shelby Thomas, Geoffrey M. Voelker, and George Porter. "CacheCloud: Towards Speed-of-light Datacenter Communication". In: *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*. Boston, MA: USENIX Association, July 2018. URL: https://www.usenix.org/conference/hotcloud18/presentation/thomas.

[76] Shelby Thomas, Rob McGuinness, Geoffrey M. Voelker, and George Porter. "Dark Packets and the End of Network Scaling". In: *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*. ANCS '18. Ithaca, New York: Association for Computing Machinery, 2018, pp. 1–14.

ISBN: 9781450359023. DOI: 10.1145/3230718.3230727. URL: https://doi.org/10.1145/3230718.3230727.

[77] *Intel® Data Direct I/O Technology (Intel® DDIO): A Primer*. Intel, Feb. 2012. URL: https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf (visited on 01/25/2022).

[78] Minhu Wang, Mingwei Xu, and Jianping Wu. "Understanding I/O Direct Cache Access Performance for End Host Networking". In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6.1 (2022), pp. 1–37.

[79] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. "Understanding host network stack overheads". In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 2021, pp. 65–77.

[80] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. "Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pp. 673–689. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/farshin.

[81] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. "Make the most out of last level cache in intel processors". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. 2019, pp. 1–17.

[82] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. "ResQ: Enabling SLOs in Network Function Virtualization". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 2018, pp. 283–297.

[83] "Packet Order Matters! Improving Application Performance by Deliberately Delaying Packets". In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022. URL: https://www.usenix.org/conference/nsdi22/presentation/ghasemirahni.

[84] *Data Plane Development Kit Power Optimization on Advantech* Network Appliance Platform*. Tech. rep. Intel, 2015. URL: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/dpdk-power-optimization-advantech-white-paper.pdf.

[85] Xuesong Li, Wenxue Cheng, Tong Zhang, Jing Xie, Fengyuan Ren, and Bailong Yang. "Power Efficient High Performance Packet I/O". In: *ACM ICPP 2018*.

[86] Sebastiano Miano, Alireza Sanaee, Fulvio Risso, Gábor Rétvári, and Gianni Antichi. "Domain Specific Run Time Optimization for Software Data Planes". In: (2022).

[87] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. "PacketMill: toward per-Core 100-Gbps networking". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2021, pp. 1–17.

[88] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. "Understanding PCIe performance for end host networking". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 2018, pp. 327–341.

[89] Roni Haecki, Lukas Humbel, Reto Achermann, David Cock, Daniel Schwyn, and Timothy Roscoe. "CleanQ: a lightweight, uniform, formally specified interface for intra-machine data transfer". In: *arXiv preprint arXiv:1911.08773* (2019).

[90] Paul E McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. "Read-copy update". In: *AUUG Conference Proceedings*. AUUG, Inc. 2001, p. 175.

[91] The kernel development community. *What is RCU? – "Read, Copy, Update"*. URL: `https://www.kernel.org/doc/html/latest/RCU/whatisRCU.html`.