Bachelor’s thesis

EFFICIENCY AND UTILIZATION OF VECTOR PACKET PROCESSING IN HIGH-SPEED NETWORKS

Ondřej Slavík

Faculty of Information Technology
Department of Computer Systems
Supervisor: Ing. Jan Fesl, Ph.D.
May 9, 2025

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Efficiency and utilization of Vector Packet Processing in high-speed networks
Student:	Ondřej Slavík
Supervisor:	Ing. Jan Fesl, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Networks and Internet 2021
Department:	Department of Computer Systems
Validity:	until the end of summer semester 2025/2026

Instructions

Vector Packet Processing (VPP) je moderní softwarový framework, který umožňuje zpracování paketů ve vysokorychlostních sítích na úrovni uživatelského prostoru operačního systému. Významnou výhodou využití VPP by mělo být výrazné zvýšení propustnosti a snížení latence v rámci vysokorychlostní sítě. Zmíněné výhody VPP jsou primárně teoretické a zatím nebyly experimentálně dostatečně prokázány.

V rámci tvorby bakalářské práce postupujte dle níže uvedených kroků:
1) Nastudujte a popište detailně všechny principy, které VPP používá, jak je implementováno a jak lze VPP efektivně využívat.
2) Vytvořte testovací scénáře, které umožní srovnat efektivitu a cenu využití VPP oproti běžnému způsobu zpracování paketů na úrovni jádra operačního systému.
3) Po poradě s vedoucím práce realizujte infrastrukturu vhodnou pro reálné otestování VPP.
4) Na základě bodu 2) proveďte dostatečný počet měření (minimálně stovky) a srovnejte možný dosažitelný průtok, latenci a spotřebu el. energie s využitím resp. bez využití VPP.
5) Proveďte důkladný rozbor a diskuzi výsledků z předchozího kroku a explicitně uveďte nevýhody využití VPP, pokud nějaké budou.

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 28 November 2024 in Prague.

Citation of this thesis: Slavík Ondřej. *Efficiency and utilization of Vector Packet Processing in high-speed networks.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2025.

I would like to express my sincere gratitude to my thesis supervisor, Ing. Jan Fesl, Ph.D., for his guidance, support, and valuable insights throughout the entire process of writing this thesis.
My thanks also go to the Silicon Hill club of the CTU Student Union for providing an inspiring environment and the technical resources that supported my work.
Finally, I would like to thank my family for their unwavering support, encouragement, and never-ending patience during my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Section 2373(2) of Act No. 89/2012 Coll., the Civil Code, as amended, I hereby grant a non-exclusive authorization (licence) to utilize this thesis, including all computer programs that are part of it or attached to it and all documentation thereof (hereinafter collectively referred to as the ”Work”), to any and all persons who wish to use the Work. Such persons are entitled to use the Work in any manner that does not diminish the value of the Work and for any purpose (including use for profit). This authorisation is unlimited in time, territory and quantity.

In Prague on May 9, 2025

Abstract

Fill in the abstract of this thesis in English. Lorem ipsum dolor sit amet. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Cras pede libero, dapibus nec, pretium sit amet, tempor quis. Sed vel lectus. Donec odio tempus molestie, porttitor ut, iaculis quis, sem. Suspendisse sagittis ultrices augue.

Keywords Vector Packet Processing, Network benchmark, Energy efficiency, Linux network stack, Data Plane Development Kit

Abstrakt

Fill in the abstract of this thesis in Czech. Lorem ipsum dolor sit amet. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Cras pede libero, dapibus nec, pretium sit amet, tempor quis. Sed vel lectus. Donec odio tempus molestie, porttitor ut, iaculis quis, sem. Suspendisse sagittis ultrices augue.

Klíčová slova enter, comma, separated, list, of, keywords, in, CZECH

Contents

List of Figures

List of Tables

List of code listings

List of abbreviations

DFA	Deterministic Finite Automaton
FA	Finite Automaton
LPS	Labelled Prüfer Sequence
NFA	Nondeterministic Finite Automaton
NPS	Numbered Prüfer Sequence
XML	Extensible Markup Language
XPath	XML Path Language
XSLT	eXtensible Stylesheet Language Transformations
W3C	World Wide Web Consortium

Introduction

Modern high-performance network devices are usually proprietary systems that combine custom hardware, specialized operating systems, and tightly coupled software. While these solutions offer high throughput and reliability, they are typically expensive, inflexible, and slower to evolve due to their closed design and development model. Vector Packet Processing (VPP) is a high-performance network stack that operates at layers 2 to 4 of the ISO/OSI model. It was originally developed by Cisco Systems, Inc. (which is a world leader in networking) and open-sourced in 2016 under the Fast Data Project (FD.io), that is part of the Linux Foundation. VPP brings the ability to perform efficient, high-speed packet processing on common off-the-shelf (COTS) hardware, across a wide range of platforms and operating systems. Its open and flexible architecture opens the door to a new class of network applications that can be deployed and scaled more easily than traditional hardware appliances. In this way, VPP could represent a shift in the traditionally conservative networking world, echoing the "Mainframe to PC" revolution, where general-purpose systems replaced proprietary platforms, enabling broader innovation and accessibility.

Since VPP was open-sourced only recently, it has not yet been widely adopted by the market, and there are only a limited number of academic studies on the subject. As a result, this area remains underexplored. This thesis aims to contribute to this field by evaluating VPP's[1] performance, with a particular focus on its electricity consumption. The findings could provide valuable insights for the industry and guide future research, especially in light of the increasing importance of energy efficiency, as highlighted in recent forecasts by ČEPS a.s. regarding the future of energy resources in the Czech Republic.

With the development of AI and the growing demand for high-resolution streaming services, it is highly likely that the demand for internet bandwidth

[1]The abbreviation VPP is also commonly used in academic literature to refer to a Virtual Power Plant.

will continue to rise. This will result in an increased need for network equipment capable of processing larger volumes of data more efficiently. Therefore, it is crucial to explore technologies like VPP that are capable to handle this growing demand and to explore their energy efficiency.

This thesis is divided into two parts: Theoretical and Practical. The Theoretical part presents the traditional approach to networking and packet processing, as well as an overview of how VPP is designed and the principles on which it operates. Additionally, it introduces the testing scenarios that were used. The Practical part describes the testing infrastructure, presents the results of various measurements, and provides an analysis of the findings.

Chapter 1
Theoretical part

1.1 Vector Packet Processing (VPP) and Its Operating Principles

This section describes the fundamental principles behind the Vector Packet Processing (VPP) technology, which aims to enable efficient and high-performance network packet processing. VPP is built on modern programming and architectural principles that allow maximum utilization of contemporary hardware, particularly in parallel processing and memory access optimization.

The section begins with a brief description of traditional network traffic processing methods used by operating systems and their limitations in terms of performance and scalability. Following that, the architecture of VPP is explored in detail, explaining how packets are processed in vectors, the use of a node graph, and the various techniques that contribute to its high efficiency—such as I/O and compute batching, zero-copy methods, and lock-free multithreading. The purpose of this section is to provide a theoretical foundation for understanding how VPP operates.

1.1.1 Traditional network traffic processing

A *network packet* is a basic unit of data transmitted over a network. It consists of a *header*, which includes control information such as source and destination IP addresses, and a *payload*, which carries the actual user data. Packets are routed independently through the network and reassembled at the destination. This structure allows efficient and reliable communication, even over complex or unreliable network paths.

Currently, packet processing works as follows: a packet arrives at the network card, which then issues a system call (syscall) to the operating system for packet processing. The microprocessor must save the currently executing instruction, perform a context switch, locate the appropriate service routine in the interrupt vector table, and handle the packet processing. Once completed, it must restore the saved instruction, perform another context switch, and return to processing the interrupted program.

This system for operating peripherals was designed under the assumption that the peripherals would not request interrupts continuously, which is not the case with network devices that need to process large volumes of data split into small parts. This method requires the microprocessor to execute a significant number of instructions not directly related to packet processing. Gallatin et al. [1] discovered that if MTU is 1500 bytes, then interrupt handling accounts for 20% - 25% of receiver packet-processing overhead. Another disadvantage of tradidtional packet processing is the inefficient handling of cache memory; the processing of the packets one by one in response to interrupts leads to frequent cache misses in both cache and inctruction caches. [2]

1.1.2 An Introduction to VPP

Vector Packet Processing (VPP) is a multi-platform network stack that operates at layers 2-4 of the ISO/OSI model and is developed by the FD.io project. It consists of a set of forwarding vertices arranged in an oriented graph and auxiliary software and provides out-of-the-box switch/router functionality. Unlike traditional network stacks, which run in the kernel, VPP operates in user space.

In a traditional approach, packets are processed one by one. In contrast, VPP reads the largest available number of packets called vector from the network interface card (NIC) and processes the entire vector through a VPP node-graph one node at a time. Each node in this graph handles a specific part of the packet processing. This approach reduces cache misses and spreads fixed overhead costs across multiple packets, lowering the average processing cost per packet. Additionally, it allows VPP to take advantage of multiple cores, enabling parallel processing, which significantly improves overall performance.

Vector Packet Processing (VPP) runs on common off-the-shelf hardware (COTS), ensuring its broad compatibility and flexibility for deployment. It supports various architectures such as x86, ARM, and Power, and can be deployed on both standard servers and embedded devices. The design of VPP is agnostic to hardware, kernel, and deployment platform, meaning it can operate across a wide range of systems, including bare metal servers, virtual machines (VMs), and containers. This approach allows VPP to be deployed on widely available infrastructure without the need for specialized hardware. [3]

1.1.3 Techniques used in VPP

According to Linguaglossa et al. [4], VPP utilizes a combination of kernel-bypass and low-level code optimization techniques to maximize packet processing efficiency and take full advantage of modern CPU microarchitectures. These techniques include:

- **Lock-Free Multi-Threading** is a programming technique that leverages modern multi-core CPUs to increase system performance. In network applications, parallelism is achieved by running multiple threads in the same time. Ideally, the more threads are used, the better the system performance but only up to a saturation point beyond which additional threads bring no gainns. However, to reach this ideal performance, traditional synchronization mechanisms such as mutexes and semaphores must be avoided, as they introduce delays due to thread contention. Instead, lock-free architectures have to be used, allowing threads to operate independently without blocking each other. In the context of VPP this approach is enabled by hardware features like multi-queue NICs, which allow each thread to handle a distinct subset of traffic, ensuring efficient and parallel processing. [4]
- **I/O batching** is a key technique used in VPP. Instead of raising an interrupt for every incoming packet, the network interface card (NIC) collects multiple packets into a buffer and triggers an interrupt only when the buffer is full. This reduces the overhead caused by frequent context switching and interrupt handling. VPP typically uses poll-mode drivers, which collect packets in batches without relying on interrupts. Moreover, the batching technique is applied system-wide in VPP. This approach maximizes CPU efficiency, improves cache usage, and delivers stable, high-throughput performance even under heavy load. [4]
- **Compute batching** is a technique that extends I/O batching to the processing phase itself. Instead of processing one packet at a time, network functions are designed to operate on entire batches of packets. This approach minimizes overhead from function calls (such as context switches and stack setup) and improves instruction cache efficiency. When a batch of packets enters a processing function, only the first packet might cause an instruction cache miss, while the rest benefit from the already warmed cache. Additionally it is possible to take advatage of instruction-level parallelism. [4]
- **Receive-Side Scaling** is a hardware-based technique used by modern NICs to distribute incoming packets across multiple RX queues. This enables parallel packet processing by allowing each queue to be handled by a separate thread, improving scalability and throughput. Packet assignment is typically done using a hash function over packet header fields (e.g., the 5-tuple). [4]

- **Zero-Copy** is a technique used to eliminate unnecessary memory copying during packet processing. Instead of copying incoming packets from the network interface card (NIC) to a separate buffer via system calls, the NIC writes packets directly into a pre-allocated memory region that is shared with the user-space application via Direct Memory Access (DMA). This allows the application to access packet data without invoking system calls or duplicating memory, which greatly reduces CPU overhead. [4]

- **Multi-loop** is a coding technique in which functions are designed to process N packets simultaneously, assuming they undergo the same operations. Because the processing of each packet is usually independent of the others, this approach enables high instruction-level parallelism and keeps CPU pipelines efficiently utilized. It requires writing explicitly parallel functions, often using C templates, and helps increase throughput by raising the number of instructions executed per clock cycle. However, its effectiveness is limited when performance is primarily constrained by memory access rather than computation. [4]

- **Data prefetching** is a technique used to preload data into the CPU cache before it is actually needed during processing. In the context of VPP, this means prefetching data for the $i+1$-th packet while the i-th packet is being processed. When combined with multi-loop processing, it is possible to prefetch data for packets $i+1$ to $i+N$ while processing packets $i-N$ to i, further improving efficiency. Although prefetching cannot be applied at the start or end of the batch (due to a lack of preceding or following packets), this limitation has negligible impact on performance because of the large batch size (usually 256 packets) typically used in VPP. The technique increases instructions per clock cycle by reducing memory access latency. [4]

- **Branch prediction** in VPP refers to a coding practice where developers provide compiler hints to indicate which branch of a conditional statement is more likely to be taken. These hints allow the compiler to generate optimized machine code that minimizes the performance cost of mispredicted branches. When the prediction is correct, the CPU pipeline continues execution without interruption, reducing wasted cycles and improving throughput. Although modern CPUs have effective built-in branch predictors, providing explicit hints can still offer performance benefits in branch-heavy code. Because the processing logic in VPP is relatively stable, such predictions are often accurate and help to improve performance. [4]

- **Function flattening** refers to the use of inline functions within VPP graph nodes to eliminate the overhead associated with standard function calls. By avoiding register shuffling and stack operations required by the Application Binary Interface, this approach reduces latency and improves execution speed. Additionally, inlining enables the compiler to perform more aggressive optimizations, such as removing unused code branches. [4]

- **Direct Cache Access** is a hardware-supported technique that allows network interface cards to write incoming packet data directly into the CPU's L3 cache, bypassing RAM. As an extension of zero-copy using DMA, this reduces memory latency and can significantly reduce RAM usage. [4]
- **Multi-architecture support** allows VPP to select the most suitable implementation of a graph node function at runtime, based on the detected CPU microarchitecture. For example, a single binary can dynamically use AVX2-optimized code on supported processors, while falling back to compatible versions on older hardware. [4]
- **Cache Coherence and Locality** are critical factors in the performance of modern software-based packet processing systems. In current COTS architectures, memory access has become a major bottleneck, which is mitigated by a multi-level cache hierarchy. Minimizing cache misses and maintaining data locality during packet processing is essential for achieving high performance and low latency. [4]

1.1.4 VPP Processing Graph and Graph nodes

At the core of VPP lies the Packet Processing Graph, a directed graph composed of relatively small, modular and loosely coupled nodes. Each node is designed to perform a specific task and there are 3 types of them: *process*, *input* and *internal*. Process nodes do not participate in the packet forwarding graph; instead, they handle timers, events, and other background tasks within the VPP runtime. Input nodes are used for input of data and internal nodes are used for vector processing. Internal nodes also serve as output nodes. When a vector of packets is prepared by input node, it is then pushed through the internal nodes. During processing, the vector may be split if the batch contains packets of different protocols or types, as they may need to follow different paths through the graph. When the original vector is completely processed, the process repeats. Illustration of this Processing Graph is shown in fig. 1.1.

Thanks to VPP's modular design, the processing graph is highly customizable and extensible. New nodes – referred to as plugins – can be easily added to implement specific functionality or repleace existing ones. Plugins are shared libraries that are loaded during startup of VPP, and they are not dependent on the VPP source code, allowing them to be developed independently. Moreover, existing nodes can be rewired to modify the packet processing logic when necessary. [4, 5, 6]

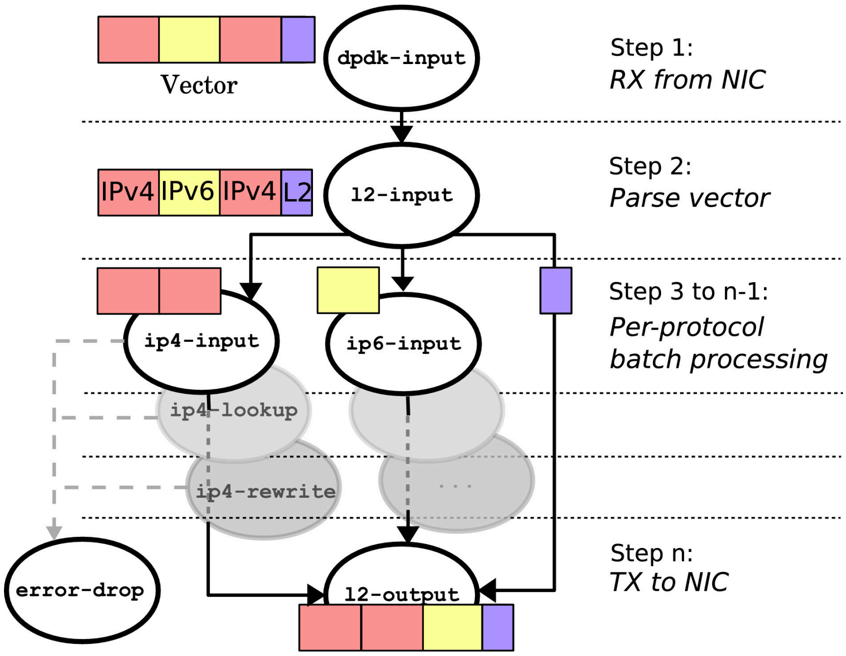

Figure 1.1 Picture showing the VPP Processing Graph [4]

1.1.5 Multithreading and Thread Roles

VPP can operate in either single-threaded or multi-threaded mode. In single-threaded mode – which is the default configuration – a single `main thread` handles all functions, including both packet processing and management tasks. In multi-threaded mode, the `main thread` is responsible for management functions (such as the debug CLI, API handling, and statistics collection), while one or more `worker threads` handle packet processing from input to output.

Each worker thread polls input queues on a subset of interfaces. When Receive Side Scaling is enabled, multiple worker threads can process different hardware queues of the same NIC in parallel. [7]

1.1.6 DPDK and Its Role in VPP

The Data Plane Development Kit (DPDK) is an open-source collection of libraries and drivers designed to support high-speed packet processing in user space. It was initially developed by Intel in 2010 and is now maintained as a Linux Foundation project. DPDK provides a set of APIs and components that allow applications to bypass the kernel network stack and to directly access network interface cards (NICs) through poll-mode drivers (PMD), significantly reducing the overhead associated with traditional packet handling mechanisms. [8]

DPDK is used in VPP for interfacing with hardware. It is implemented as a plugin called *dpdk-plugin*. [4, 5]

While VPP supports multiple mechanisms for accessing network devices, such as *af_packet*, to the best of the author's knowledge, DPDK is by far the most widely used option.

1.1.6.1 Poll Mode Drivers

Poll Mode Drivers (PMDs) are a key component of the DPDK framework. Unlike traditional network drivers, which rely on interrupts to signal packet arrival, PMDs continuously poll the network interface card (NIC) – specifically its RX queue – in a busy-loop, completely avoiding traditional interrupt-based mechanisms. This approach allows packets to be retrieved, processed, and delivered directly to user space without kernel involvement. While this results in very low latency and high throughput, it also causes constant CPU utilization on the cores assigned to polling, regardless of the traffic load. [9]

Not every network interface card is supported by DPDK. Each supported device requires a specific Poll Mode Driver (PMD), which must be available and compatible with the given hardware. An up-to-date list of supported NICs and their corresponding PMDs is maintained on the official DPDK website. [10]

1.1.6.2 Memory management and Hugepages

DPDK uses a user-space memory model that eliminates the need for kernel involvement during packet processing. It operates on memory regions reserved as hugepages – large memory pages, typically 2 MB or 1 GB in size, which are allocated at startup. These hugepages are used to store packet buffers and manage memory pools. DPDK defines its own memory management structures, such as mempools, which consist of preallocated fixed-size objects.

DPDK is also explicitly NUMA-aware. Most memory allocation functions require the application to specify the target NUMA node, ensuring that memory is allocated close to the CPU core accessing it. This minimizes latency caused by cross-node memory access and helps optimize performance on multi-socket systems. [11]

1.1.6.3 Packet Reception and Transmission: A comparison between Linux Network Stack and DPDK

When a packet arrives at a NIC managed by the Linux Network Stack, it is first stored in the NIC's internal buffers. The NIC then writes the packet via Direct Memory Access (DMA) to the section of RAM provided by the driver and updates the corresponding descriptor in the RX buffer. The RX buffer is implemented as a ring queue.

Once the packet has been saved, the corresponding interrupt request (IRQ) is triggered to notify the CPU that one or more packets have arrived in that queue. Then, the corresponding IRQ handler is executed, which acknowledges the interrupt and calls the *napi_schedule* and *___raise_softirq_irqoff* functions.

The first function marks the associated `napi_struct`[1] as ready for processing, while the second one raises a software interrupt (SoftIQR) specifically intended for processing incoming packets. Once the SoftIRQ is triggered, the kernel handles the actual packet processing in a deferred context. It goes through a list of network devices that have indicated pending work (i.e., their associated `napi_struct` has been marked as ready to be processed). and calls their associated poll functions to retrieve and process packets from the receive queues.

This happens on the same CPU core that handled the original interrupt. If the system is busy or the processing takes too long, the remaining work may be handled by the *ksoftirqd* kernel thread. The packets may be aggregated into a single larger packet using Generic Receive Offload (GRO), or processed individually. In both cases, they are passed to the IP stack via the `netif_receive_skb` function.

The transmission path is handled in a similar manner, using ring buffers, DMA, and deferred processing. However, unlike reception, packet transmission is initiated from the IP stack using the `__dev_queue_xmit` function. Depending on the qdisc in use, packets are either enqueued in the software queue or passed directly to the driver for transmission. Once a packet is selected for transmission, the driver places a descriptor into the TX ring buffer and sets up DMA so that the NIC can read the packet data from memory. After the NIC finishes transmitting the packet, it triggers a TX interrupt, which allows the driver to perform post-processing such as unmapping DMA buffers and freeing memory. [12]

When there is a NIC with multiple RX queues available, it is assigned to one of the queues based on the NIC's configuration[2]. The selection of the target queue is typically based on a hash function computed over network and/or transport layer headers. Each queue has a dedicated IRQ, which can be assigned to specific CPU cores based on system settings. This mechanism is known as Receive Side Scaling (RSS).

When sending a packet from a NIC equipped with multiple TX queues, Transmit Packet Steering (XPS) is used to determine the appropriate TX queue. The first option is that a CPU core is assigned specific TX queues. The other option is to use the TX queue corresponding to the RX queue from which the flow originated. If multiple queues are eligible, a hash function is used to select the specific queue. [13]

In comparison, incoming packets in DPDK are delivered by the network

[1] `napi_struct` represents a NAPI context associated with a specific receive queue of a network device.

[2] For network interface cards with multi-queue capabilities, the corresponding kernel driver often provides a module parameter to define how many hardware queues should be initialized and utilized.

interface card (NIC) using direct memory access (DMA), which writes packet data into pre-allocated memory buffers specified by receive (Rx) descriptors. These descriptors are organized in a circular ring (Rx queue), where the NIC populates entries at the head, while the VPP continuously polls the tail using functions such as *rte_eth_rx_burst()*. This polling mechanism enables the VPP to retrieve multiple packets in a batch, minimizing interrupt overhead and reducing latency, thereby increasing throughput and core efficiency. [14]

Transmission is handled similarly, using a ring buffer known as the transmit (Tx) queue. The application prepares transmit (Tx) descriptors at the tail of this queue, each containing the address and length of the packet to be sent. These descriptors reference memory buffers (mbufs) holding the packet data. After the descriptors are written, the application updates the Tx queue's tail pointer to notify the NIC that new packets are available. The NIC then reads the descriptors from the head of the queue, fetches the packet data via DMA, and transmits the packets on the wire. [15]

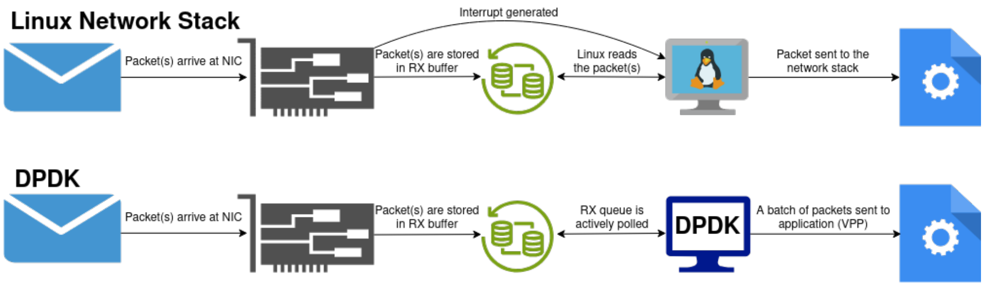

Figure 1.2 Diagram illustrating the differences in packet handling between the Linux Network Stack and DPDK.

Based on the description above, several key differences between the Linux Network Stack and DPDK can be observed. Although both rely on a similar underlying mechanism – ring buffer queues – their implementations differ fundamentally. In the Linux Network Stack, memory management is handled by the kernel through device drivers. In contrast, DPDK allocates memory in user space and manages it through its own framework, providing packet buffers and ring structures directly to the application.

Linux is heavily dependent on hardware interrupts (IRQs) for packet reception and software interrupts (softIRQs) for deferred processing, which introduces frequent context switches. While NAPI uses polling to process packets from receive queues, the packets are still handled by kernel strictly one by one. This increases the likelihood of cache misses during packet processing, as each packet is processed independently and may not benefit from cache locality.

In contrast, DPDK works entirely in user space and uses continuous active polling, completely bypassing the need for interrupts and context switching. Since it does not wait for an interrupt to occur, packet processing can begin sooner, reducing initial latency. Additionally, DPDK can retrieve multiple packets in a single burst, preparing them for vectorized processing in VPP. Figure 1.2 presents a simplified diagram highlighting the key differences.

1.2 Implementation of Vector Packet Processing

VPP's dataplane is implemented by four main architectural layers: VPPINFRA, VNET, VLIB, and Plugins. Each of these layers provides distinct functionality that supports efficient networking operations, from low-level data structure management to high-level network function optimizations.

VPPINFRA provides foundational libraries for tasks such as memory handling, vectors, rings, hash table lookups, and timers. VNET focuses on implementing network protocols for layers 2 to 4 and includes the control plane. VLIB serves as the runtime environment for vectorized processing and also provides the command-line interface. Finally, plugins allow the system to be extended or customized by adding new features or modifying existing ones [16].

1.2.1 VPPINFRA

VPPINFRA is a collection of foundational libraries designed to provide high-performance capabilities for various internal tasks within VPP. It includes dynamic arrays, hash tables, bitmaps, timing utilities, logging mechanisms, and data structure serialization, all optimized for speed and efficiency. [17]

- **Vectors** –- dynamically resized arrays with user-defined headers. Vectors are used as the basis for other structures such as pools or hash tables and support efficient memory reuse through safe length resetting. [17]

- **Bitmaps** -– compact data structures used to efficiently track the true/false state of multiple indexed items using individual bits, built on top of vectors. [17]

- **Pools** –- structures used to quickly allocate and free fixed-size data structures, such as packet buffers or per-session metadata. Internally, they are implemented using vectors and bitmaps. [17]

- **Hashes** – lookup structures optimized for fast access using hash functions. [17]

- **Timekeeping** – utilities providing precise, low-overhead timing based on CPU cycles. VPPINFRA continuously adjusts its time calibration by comparing CPU ticks against kernel time, ensuring accurate time measurement without expensive system calls. [17]

- **Timer wheel** – subsystem for efficiently managing timers and periodic events. It supports multiple configuration options, including the number of wheels, slots, and timers per object, allowing high-performance scheduling in time-sensitive applications. [17]

- **Logging and formatting** – includes support for fast event logging, trace output, and data formatting used for debugging and diagnostics. [17]
- **Serialization** – support for serializing and deserializing internal data structures for persistent storage or communication between threads. [17]

1.2.2 VNET

VNET (VPP Network Stack) implements the core networking logic in VPP, providing graph nodes for Layer 2 and Layer 3 packet processing.

A key mechanism provided by VNET is the concept of feature arcs. These represent named sequences of graph nodes within the packet processing graph, allowing custom nodes – such as NAT, ACLs, or telemetry – to be inserted into existing pipelines in a defined order. Feature arcs enable modular composition of processing features without modifying the core graph logic. For example, an ACL node can be inserted at the beginning of the `ip4-unicast`[3] feature arc.

In addition to protocol and interface handling, VNET also provides a flexible framework for packet tracing, allowing developers to inspect and debug the path that each packet takes through the graph in fine detail. This is especially useful for analyzing the behavior of custom nodes or diagnosing complex feature interactions.

Finally, VNET includes a built-in packet generator, which can be used to simulate traffic and evaluate the performance of specific graph paths under controlled conditions. [18, 19]

1.2.3 VLIB

VLIB provides the runtime environment and execution engine that powers VPP's packet processing model. One of its core responsibilities is managing the registration and execution of all graph nodes, including input, internal and process nodes.

The execution of graph nodes in VPP is coordinated by a lightweight cooperative scheduler. Each iteration of the main loop begins with input nodes producing vectors of packets, which are then passed through a sequence of internal nodes forming a directed graph. Nodes process the incoming vector and determine, for each packet, which next node should process it based on routing, classification, or protocol-specific logic.

Packets destined for the same next node are grouped together and placed into a new `vlib_frame_t`, which is then enqueued to that node for processing.

[3] `ip4-unicast` is a feature arc that processes unicast IPv4 packets before they reach the routing logic.

This selective forwarding enables efficient vector splitting and maintains high performance by improving cache locality and reducing per-packet overhead.

When a node cannot or should not be executed immediately, VPP defers its execution by adding it to a list of pending operations using `pending_frames`. These frames are processed later in the main loop.

VLIB also provides the command-line interface (CLI), which allows operators to interact with the VPP runtime. [20]

1.2.4 Plugins

Plugins in VPP are implemented as shared object libraries that are automatically discovered and loaded by VLIB during startup. They allow developers to add new features or extend existing functionality without modifying the VPP core.

To create a plugin, developers add a new directory under `src/plugins`, define build instructions using `plugin.mk` and `CMakeLists.txt`, and implement the required logic. After compilation, the plugin is placed into the designated plugin directory and becomes available for VPP to load.

This modular architecture enables rapid experimentation and integration of custom network functions while keeping the base system clean and maintainable. [21, 22]

1.2.5 Installation, Configuration and Startup

VPP officially supports the *x86_64* and *ARM-AArch64* architectures and can be installed on recent LTS versions of Debian and Ubuntu Linux distributions. [23] Prebuilt packages for these systems are provided via the official FD.io package repository hosted on Packagecloud.io [24]. Alternatively, VPP can be built from source code to enable custom builds, which can be useful for development purposes. This approach also enables installation on distributions such as Red Hat and CentOS, which are not officially supported by the prebuilt packages. [25].

1.2.5.1 System Preparation

After installation, VPP registers itself as a system service and a corresponding systemd unit file is created at `/usr/lib/systemd/system/vpp.service`. This enables standard service management commands such as `systemctl start vpp` or `systemctl status vpp` to control the daemon. The service uses the default configuration defined in `/etc/vpp/startup.conf`, unless otherwise specified.

Since VPP requires hugepages, the installation process places a configuration file at `/etc/sysctl.d/80-vpp.conf` to override system defaults. By default, this configuration sets the number of 2 MB hugepages to 1024.[4] [26]

Before running VPP with DPDK, the relevant network interfaces must either be detached from their kernel drivers and bound to a kernel module providing user-space access, such as `vfio-pci`, or used with a bifurcated driver[5]. In the former case, the `dpdk-devbind.py` utility can be used to rebind interfaces as needed. This setup is required for DPDK to successfully initialize the specified network interface. [27]

1.2.5.2 Startup Configuration

Although VPP can be started with command-line arguments, such as `/usr/bin/vpp unix { interact` which can be useful for debugging purposes, it is typically started using a configuration file. The default configuration file is located at `/etc/vpp/startup.conf`, but a custom file can also be specified. When not using `systemd`, a specific configuration file can be passed manually via the `-c` parameter, for example: `/usr/bin/vpp -c /etc/vpp/startup.conf`. [28]

The configuration file is divided into several sections. The following overview summarizes the most relevant ones, as described in the official configuration reference. [29]

- **The unix section** defines the general runtime behavior of the VPP process. It includes parameters that control how the application runs, where it logs output, and how the command-line interface is exposed. The `nodaemon` parameter prevents VPP from running in the background. The `log` parameter specifies the path to the log file, typically set to `/var/log/vpp/vpp.log`. The `cli-listen` parameter defines how the VPP CLI is made accessible. A local UNIX socket or pair of ip address and port can be used. Additionally, the `startup-config` parameter (or its alias `exec`) can be used to provide a file containing CLI commands that are executed automatically after VPP startup. [29]
- **The api-trace section** controls how API message tracing is handled within VPP. This section is primarily used for debugging and development purposes. API message tracing is typically enabled using the `api-trace on` command. [29]
- **The cpu section** is used to configure how VPP threads are mapped to CPU cores. The `main-core` parameter specifies the core on which the main thread runs. The `corelist-workers` parameter defines a list of cores used by worker threads responsible for packet processing. The `scheduler-policy` parameter allows selecting a thread scheduling strategy, such as `fifo`, `rr`, or others, depending on the desired behavior. [29]

---

[4] This is a system-wide setting, not specific to VPP.

[5] In this case, the NIC remains under kernel control, while the data path is handled by the PMD.

- **The buffers section** allows configuring the size and number of memory buffers used by VPP for packet processing. [29]
- **The dpdk section** configures how DPDK is initialized and used by VPP. The `dev` parameter specifies the PCI address of a network interface to be managed by VPP/DPDK. The `uio-driver` parameter sets the kernel module providing user-space access to be used, commonly `vfio-pci`. The `num-mbufs` parameter defines the number of memory buffers allocated for packet processing. Additional options include parameters such as `num-rx-queues` and `num-tx-queues`, which specify the number of receive and transmit queues for each device. [29]
- **The plugins section** allows enabling or disabling individual VPP plugins during startup. Each plugin can be explicitly marked as `enable` or `disable`, depending on whether it should be loaded. It is also possible to disable all plugins by default. [29]

Other configuration sections exist for advanced or highly specific use cases, such as memory tuning, internal event logging, or telemetry export, and are, to the author's best knowledge, used only in fine-tuned or specialized deployments.

1.3 Utilization of Vector Packet Processing

VPP supports a comprehensive set of Layer 2 to Layer 4 network functions. At Layer 2, it provides Ethernet bridging, MAC learning, VLAN tagging (including dot1q and QinQ), and support for L2 cross-connects and policers.

At Layer 3, VPP implements both IPv4 and IPv6 routing with ECMP support, NAT44/NAT64, and ACL-based filtering. It also supports tunneling mechanisms such as GTP-U, IP-in-IP, and VXLAN. Segment routing (SRv6), LISP, and punt redirect mechanisms are included as well.

At the transport layer (L4), basic UDP and TCP stack functionality is available.

Additionally, supported features include PPPoE, the WireGuard VPN protocol, GRE tunneling, DHCP client and proxy functionality, and L2TPv3.[30]

According to the VPP's authors [3], VPP can be for example effectively utilized as a virtual switch, virtual router, gateway or used as a basis for a firewall, IDS and load balancer. It already includes enough features to be deployed in production environments.

1.3.1 Integration with the SDN/NFV Ecosystem

To meet the requirements of modern virtualized and cloud-native networking environments, Vector Packet Processing (VPP) was architected with a clear separation between the data plane and control plane. This design choice enables its integration into SDN and NFV frameworks, where packet forwarding logic can operate independently from centralized control mechanisms. VPP's modularity and userspace implementation allow it to function efficiently within dynamic, multi-tenant infrastructure, while remaining compatible with orchestration systems and control-plane protocols commonly used in such deployments

VPP is fully compatible with both Virtual Network Functions (VNFs) and Cloud-Native Network Functions (CNFs). Its modular architecture allows deployment in environments utilizing service function chaining, Kubernetes-based orchestration, or OpenStack-based infrastructures. Because of its userspace design and performance-optimized data plane, VPP can serve as the fast packet processing backend for SDN-controlled systems and NFV orchestrators.[31]

1.3.2 VPP as a Complete Router Solution

VPP is implemented solely as a data-plane, meaning it is not a complete routing solution on its own. VPP is dedicated to efficiently forwarding packets between interfaces based on routing rules and access control filters, but it does not include a native control-plane or support for dynamic routing protocols such as BGP or OSPF.

However, as demonstrated by the authors of the VBSR (VPP-Bird Software Router) project [32], it is possible to integrate VPP with additional components such as the Linux Control Plane (Linux-CP) plugin and the BIRD routing daemon. Bird acting as a control-plane enables dynamic routing using protocols like BGP and the Linux-CP is responsible for communication between VPP and BIRD. This integrated system creates a nearly feature-complete router solution, comparable in functionality to commercial routers.

It is important to note, however, that firewall functionality is still limited and was left by authors of VBSR as a future work. [32] While VPP supports basic packet filtering through ACLs, it lacks advanced stateful firewall features [30]. These would need to be handled externally.

1.4 Survey of Traffic Generation Tools

In order to evaluate the performance of network devices and data-plane frameworks such as VPP, synthetic traffic must be generated in a controlled and reproducible manner. Selecting appropriate traffic generation tools is therefore essential for conducting accurate benchmarking and stress-testing. Although numerous traffic generation tools exist [33], this section focuses on a subset commonly used for high-performance benchmarking and synthetic traffic generation in research and practice, namely iPerf3, D-ITG, TRex, Pktgen-DPDK & Genesids.

- **iPerf3** – iPerf3 is a network testing tool used to measure TCP, UDP, and SCTP throughput between two endpoints. It allows detailed configuration of testing parameters such as buffer size, number of parallel streams, test duration, and jitter. iPerf3 can also measure jitter, providing insights into the variation in packet arrival times, which is useful for evaluating network stability. Its client-server architecture makes it a common tool for performance benchmarking of networks and devices. [34]
- **D-ITG** – Distributed Internet Traffic Generator is a network traffic generator designed to produce traffic flows that accurately emulate a wide range of real-world application behaviors. It supports multiple transport layer protocols, including TCP, UDP, DCCP, and SCTP. D-ITG allows users to define parameters such as packet size, inter-departure time, and number of flows, making it suitable for controlled experiments on delay, jitter, packet loss, and throughput. It can operate in both single-node and distributed modes, enabling flexible deployment for testing complex topologies and performance conditions. D-ITG also includes tools for logging and analyzing the generated traffic, facilitating detailed post-experiment evaluation. [35]
- **TRex** – TRex, developed by Cisco, is a high-performance, stateful and stateless traffic generator built on top of DPDK. It supports the generation of realistic Layer 4–7 traffic using pre-recorded PCAP files and emulates multiple concurrent users and flows. TRex is especially suited for benchmarking network function virtualization (NFV) platforms, routers, and firewalls in both laboratory and production-like environments. [36]
- **Pktgen-DPDK** – Pktgen-DPDK is a high-performance traffic generator tool developed as part of the Data Plane Development Kit (DPDK). Pktgen-DPDK supports various network protocols, including IPv4, IPv6, UDP, and TCP. The tool allows precise control over traffic parameters, such as packet rate, size, and timing. Pktgen-DPDK is used in network performance tests and can capture packet-level statistics to assess the performance of the devices under test. [37]

Among the reviewed tools, the author decided to utilize iPerf3 and TRex in the subsequent experimental evaluation. IPerf3 was selected due to its status as a de facto standard for basic throughput and jitter measurements, ease of use, and widespread adoption in academic and practical contexts. TRex was chosen for its modern architecture, support for high-speed stateful and stateless traffic generation, and ability to simulate real-world traffic. In addition, TRex provides a Python-based API that enables scripting and automation of test scenarios, making it well-suited for integration into continuous testing pipelines and reproducible experiments. Other tools, such as Pktgen-DPDK and D-ITG, were excluded due to their relatively complex usage (Pktgen-DPDK) or limited maintenance and outdated design (D-ITG).

Chapter 2

Pratical part

2.1 Building Infrastructure for Measurement

The testing infrastructure has been implemented as recomended in RFC 2544 [38], which defines methods for evaluating network performance. It consists of a device under test (DUT), connected to a measurement device called *Tester*[1]. In line with the more modern RFC 8219, which states that: "All tests described SHOULD be performed with bidirectional traffic" [39], the infrastructure is designed to operate with bidirectional traffic. This approach ensures more accurate performance measurement under real-world network conditions, as opposed to unidirectional traffic. The Device Under Test (DUT) and the measurement device are connected using 100Gbit capable cables, preventing any potential bottlenecks in the connection. The illustration of this hardware setup is shown in fig. 2.1

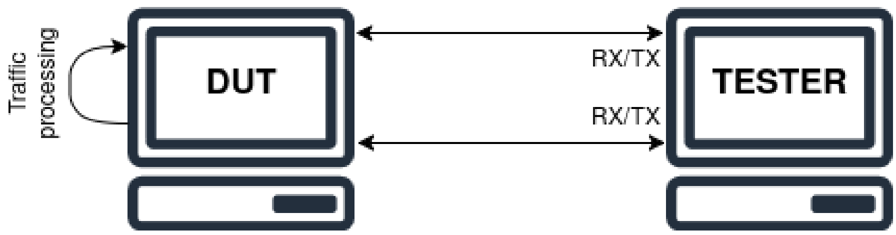

Figure 2.1 Picture showing hardware setup

The Device Under Test (DUT) is the network device being evaluated during testing. It is configured with a specific network stack and settings based on measurement scenario and serves as the focus of performance and behavior analysis in a controlled test environment. The DUT is responsible for processing network traffic and responding to the test conditions set by the measure-

[1]The hardware used in this testing setup was loaned free of charge for the purposes of this bachelor thesis by Silicon Hill club.

ment device. Additionally, the electrical power consumption of the DUT is monitored and measured during the tests to assess its energy efficiency under varying loads. The hardware of DUT is shown in table 2.1.

Table 2.1 Hardware details for DUT (Device Under Test)

Hardware Component	**DUT (Device Under Test)**
CPU Model	2x Intel(R) Xeon(R) CPU E5-2660 v3
Frequency	2.60GHz
Cores	10 physical cores each (one thread per core)
Memory (RAM)	Size, type, speed
Network Interface Cards (NIC)	Mellanox ConnectX-6 Dx (Dual-port)

The Tester (Measurement Device), on the other hand, is responsible in generating the network traffic and capturing the responses from the DUT. Its physical features are shown in table 2.2.

Table 2.2 Hardware details for Tester (Measurement Device)

Hardware Component	**Tester (Measurement Device)**
CPU Model	2x Intel(R) Xeon(R) Gold 6136 CPU
Frequency	3.00GHz
Cores	12 physical cores each (two threads per core)
Memory (RAM)	Size, type, speed
Network Interface Cards (NIC)	2x Mellanox ConnectX-5

The DUT is running Debian GNU/Linux 12 (Bookworm) x86_64 with Linux kernel version *6.1.0-32-amd64*, VPP v25.02-release, and DPDK version 24.11.1.

The tester is running ...

2.2 Metodology

The RFC 2544 recommends that each test be at least 60 seconds in duration [38]. In this work, the duration was extended to 120 seconds and each test was repeated 20 times to ensure greater stability and statistical relevance of the results. The reported values represent the arithmetic mean of these 20 measurements. In cases where an error, anomalous spike or irregularity was observed in the results, the corresponding measurement was discarded and the test was repeated. All these steps were taken to ensure consistency and statistical reliability.

The following metrics were collected: the number of transmitted and received packets and bytes; average, minimum, and maximum one-way latency; jitter; and total energy consumption of the DUT, expressed in watt-hours.

All numerical results are rounded to two decimal places. Since traffic generation was performed using TRex, its potential measurement inaccuracy must be taken into account when interpreting the results. To evaluate energy efficiency, the number of packets per watt-hour (PPWh) and bytes per watt-hour (BPWh) was used, considering only successfully delivered packets. Power consumption was measured using a Raritan PX3-5498-K1 unit running firmware version 4.2.0.5-50274. The idle power consumption of the DUT is approximately 5 Wh per two minutes.

2.3 Test Scenarios & Results

To provide a comprehensive and representative view, the tests are structured into five subsections, each corresponding to a different Ethernet frame size. Four of the selected sizes – 64 bytes, 512 bytes, 1280 bytes, and 1518 bytes – are recommended by RFC 2544 [38] covering both edge cases and practically relevant intermediate values. The fifth size, 889 bytes, was chosen based on real-world traffic analysis by Jurkiewicz et al. [40], who identified it as the average frame size observed in modern network environment. This selection covers the full range of standard Ethernet frame sizes, from the minimum to the maximum non-jumbo frames, while also including a statistically representative average.

All tests were conducted at four different transmission speeds – 1, 10, 25, and 40 Gbit/s – to evaluate the behavior of each configuration under varying network loads.

Traffic in all scenarios is generated using TRex with the TBD profile, which ensures that each packet carries a unique source IP address to simulate multiple concurrent clients, while maintaining a single destination IP per direction. Since the aim of this thesis is to evaluate the VPP architecture rather than specific features (e.g., routing table lookup or hashing mechanisms), the routing table of the DUT contains only two active forwarding entries corresponding to the test routes, along with two administrative entries used for management purposes.

The DUT is configured with the VPP stack and tested under three levels of parallelism: using 1, 4, and 10 worker threads, plus a single main thread in all configurations. The worker threads are pinned to the NUMA node closest to the NICs to minimize memory access latency. The number of RX/TX queues is aligned with the number of active worker threads in each configuration to ensure balanced packet distribution and optimal resource utilization.

To provide a baseline for comparison, all scenarios are also executed using the standard Linux kernel networking stack. It is configured with routing and interface parameters equivalent to the VPP setup, utilizing all 10 CPU cores on the NUMA node closest to the NICs. RPS is enabled, with affinities set evenly across the cores. This allows for a direct comparison between VPP and traditional kernel-based forwarding in terms of performance and energy efficiency.

2.3.1 One-way forwarding

These tests were conducted in a one-way configuration, as suggested by RFC2544[38]. This scenario can simulate networks with asymmetric traffic patterns (e.g., an HTTP server) or reflect conditions similar to a DoS attack.

2.3.1.1 1 Gbps Test Results

Table 2.3 Results of one-way 1 Gbit/s of 64-byte frames

Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
VPP-1	5.68	0.00	12.8	9.5
VPP-4	6.46	0.00	27.45	13.55
VPP-10	7.86	0.00	28.3	12.65
Linux	6.78	0.00	108.05	97.25

Table 2.4 Results of one-way 1 Gbit/s of 512-byte frames

Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
VPP-1	5,69	0.00	12.1	11.7
VPP-4	6.48	0.00	22.85	17.4
VPP-10	TODO			
Linux	6.23	0.00	56.1	51.35

Table 2.5 Results of one-way 1 Gbit/s of 889-byte frames

Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
VPP-1	5,76	0.00	10.3	8.4
VPP-4	6.45	0.00	21.3	17.6
VPP-10	7.85	0.00	20.95	16.3
Linux	TODO			

Table 2.6 Results of one-way 1 Gbit/s of 1280-byte frames

Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
VPP-1	TODO			
VPP-4	6.46	0.00	18.3	17.2
VPP-10	7.84	0.00	18.8	15.95
Linux	TODO			

Table 2.7 Results of one-way 1 Gbit/s of 1518-byte frames

Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
VPP-1	5.66	0.00	7.2	5.5
VPP-4	6.42	0.00	16.1	15.75
VPP-10	7.84	0.00	18.8	15.95
Linux	TODO			

2.3.1.2 10 Gbps Test Results

As shown in Tab. 2.8, both VPP-1 and the Linux stack were unable to handle this packet rate at 10 Gbit/s with 64-byte frames, resulting in high packet loss and significantly increased latency. VPP-4 managed to process the traffic with minimal loss, achieving the lowest latency and jitter among all configurations. VPP-10 also completed the test losslessly, but with slightly higher latency.

Table 2.8 Results of one-way 10 Gbit/s of 64-byte frames

Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
VPP-1	5.72	59.5	589.50	14.05
VPP-4	6.56	0.02	20.60	6.00
VPP-10	8.04	0.00	30.00	12.90
Linux	7.35	80.97	3846.30	217.8

It can be seen from Tab. 2.9 that increasing the frame size to 512 bytes allowed all tested configurations to handle the traffic without packet loss. Although the latency of the Linux stack improved significantly compared to the previous measurement, it still remained the highest among all configurations. The energy consumption of the Linux configuration decreased, likely due to reduced overhead from system calls.

Table 2.9 Results of one-way 10 Gbit/s of 512-byte frames

Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
VPP-1	5.60	0.00	19.95	12.00
VPP-4	6.48	0.00	28.20	14.50
VPP-10	7.97	0.00	29.05	13.65
Linux	6.85	0.00	129.4	99.35

In this test with 889-byte frames, all VPP configurations show similar results to the previous measurement, although with slightly increased jitter. This may be due to VPP constructing processing vectors based on the number of packets rather than their total size. The increased frame size improved Linux's average latency, possibly due to reduced per-packet processing overhead. The discussed results are summarized in Table 2.10.

Table 2.10 Results of one-way 10 Gbit/s of 889-byte frames

Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
VPP-1	5.62	0.00	23.4	14.95
VPP-4	6.48	0.00	26.5	18.25
VPP-10	7.95	0.00	26.95	17.55
Linux	6.67	0.00	66.45	68.3

As shown in Table 2.11, the VPP configurations performed similarly to the previous measurement. In contrast, Linux achieved slightly improved latency and jitter, potentially due to the larger frame size further reducing per-packet processing overhead.

Table 2.11 Results of one-way 10 Gbit/s of 1280-byte frames

Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
VPP-1	5.58	0.00	22.8	17.05
VPP-4	6.47	0.00	25.95	16.90
VPP-10	7.98	0.00	26.80	16.60
Linux	6.54	0.00	57.05	65.50

The results for 1518-byte frames are consistent with those observed at 1280 bytes, with no significant improvements in consumption, latency, or jitter. This suggests that increasing the frame size beyond 1280 bytes does not bring further performance gains in the tested configurations. The detailed results are presented in Table 2.12.

Table 2.12 Results of one-way 10 Gbit/s of 1518-byte frames

Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
VPP-1	5.57	0.00	20.45	14.95
VPP-4	6.48	0.00	24.00	17.45
VPP-10	7.97	0.00	25.40	17.70
Linux	6.51	0.00	61.75	65.80

Figure 2.2 shows the energy efficiency of each configuration in this test in terms of delivered packets and bytes. The significant drop in performance for VPP-1 and Linux in 64-byte frames test is caused by large packet loss. When all packets are successfully delivered, all VPP configurations maintain stable BPWh values, which is due to their busy-wait processing model. The Linux stack, on the other hand, becomes more efficient with increasing frame size, likely as a result of less frequent system calls.

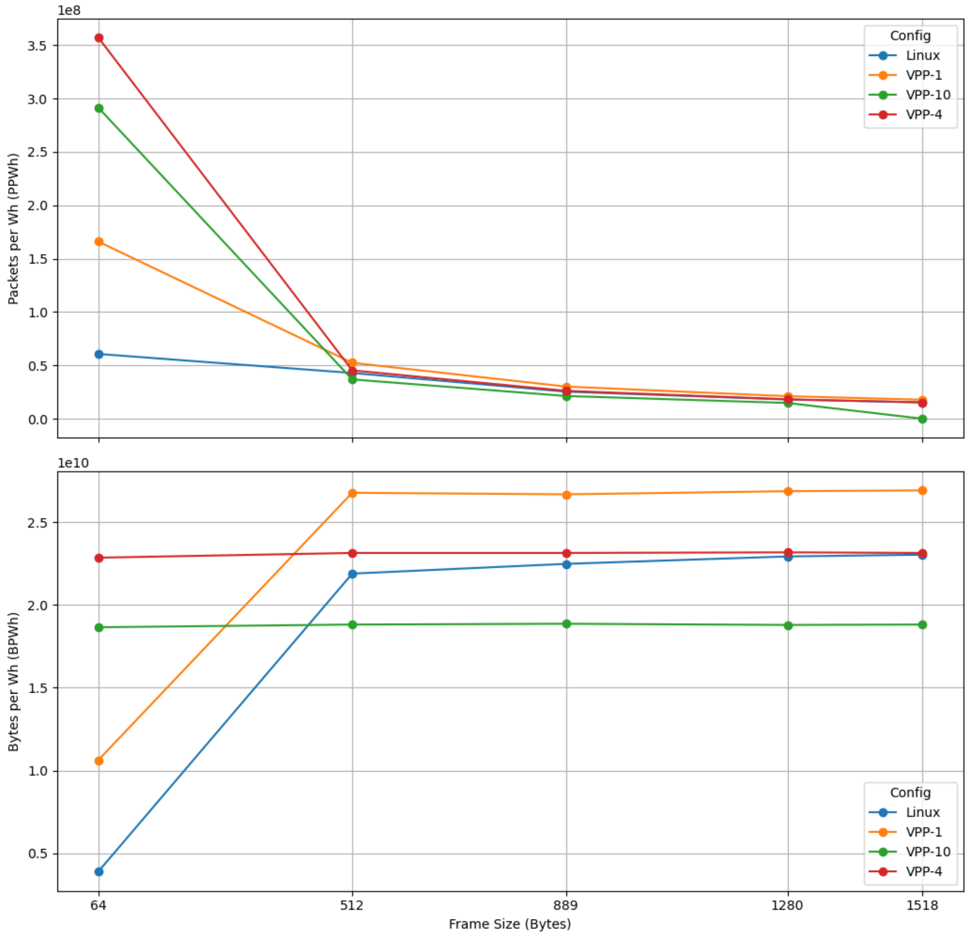

Figure 2.2 Energy efficiency per delivered data in one-way 10 Gbit/s.

2.3.1.3 25 Gbps Test Results

As shown in Table 2.13, none of the configurations were able to deliver all data. In all VPP setups, the delivered data maintained low jitter, likely due to lost packets in RX queues being overwritten by fresher incoming traffic. This stress test also demonstrates that the Linux stack is unable to handle such network load, resulting in extremely high latency of delivered packets. These results highlight the advantages of VPP under heavy traffic conditions.

Table 2.13 Results of one-way 25 Gbit/s of 64-byte frames

Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
VPP-1	5.59	83.29	577.5	7.70
VPP-4	6.56	49.00	198.85	7.65
VPP-10	8.26	29.55	155.35	11.95
Linux	7.43	92.71	5597.95	632.00

When the frame size increased to 512 bytes, all VPP configurations were able to deliver the full traffic, except for VPP-1, which dropped a negligible amount of packets. All VPP setups showed reduced latency and jitter compared to the previous test. The Linux stack, however, was still unable to handle the traffic, dropping nearly 50% of the packets and showing extremely high latency in the delivered traffic, as summarized in Table 2.14.

Table 2.14 Results of one-way 25 Gbit/s of 512-byte frames

Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
VPP-1	5.57	0.07	31.85	11.15
VPP-4	6.43	0.00	29.55	13.75
VPP-10	8.03	0.00	31.00	14.45
Linux	7.57	47.97	7819.60	477.55

As indicated by the results in Table 2.15, the VPP-1 configuration achieved lower latency, likely due to the absence of packet drops. Even when an average packet size was used, the Linux stack was still unable to deliver all packets, with measurable losses. This likely contributed to the relatively high latency and jitter observed in the Linux configuration.

Table 2.15 Results of one-way 25 Gbit/s of 889-byte frames

Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
VPP-1	5.67	0.00	23.45	11.15
VPP-4	6.39	0.00	30.50	15.50
VPP-10	8.01	0.00	29.45	15.55
Linux	7.42	0.87	166.05	111.15

In this test, using 1280-byte frames, all configurations were able to deliver the complete dataset. Without exception, all VPP configurations achieved results comparable to the previous measurement. The use of larger frames had a positive impact on latency in the Linux configuration, as indicated by the results in Table 2.16.

Table 2.16 Results of one-way 25 Gbit/s of 1280-byte frames

Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
VPP-1	5.68	0.00	23.85	10.30
VPP-4	6.48	0.00	31.15	15.40
VPP-10	7.98	0.00	28.40	16.15
Linux	7.10	0.00	146.90	126.20

As indicated by the results in Table 2.17, increasing the frame size to the maximum had no impact on performance compared to the previous measurement. Only the Linux stack showed slightly better results in terms of latency and jitter.

Table 2.17 Results of one-way 25 Gbit/s of 1518-byte frames

Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
VPP-1	5.70	0.00	24.45	14.90
VPP-4	6.46	0.00	30.60	15.80
VPP-10	7.96	0.00	28.9	15.65
Linux	7.00	0.00	130.00	105.30

Figure 2.3 illustrates the energy efficiency of each configuration in terms of delivered packets and bytes. Compared to the previous test, the Linux stack performed significantly worse, while VPP maintained stable BPWh, except in the case of 64-byte frames.

2.3.1.4 40 Gbps Test Results

The results of the 40Gbit/s test, summarized in Table2.18, clearly show that all configurations struggled to process the traffic. The performance was comparable to the 25 Gbit/s test with 64-byte frames, but with even higher packet loss rates – with the Linux stack delivering less than 5% of the packets.

Table 2.18 Results of one-way 40 Gbit/s of 64-byte frames

Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
VPP-1	5.60	89.53	576.00	6.50
VPP-4	6.57	67.54	195.70	6.57
VPP-10	8.12	54.38	152.05	9.20
Linux	7.34	95.43	5629.05	550.30

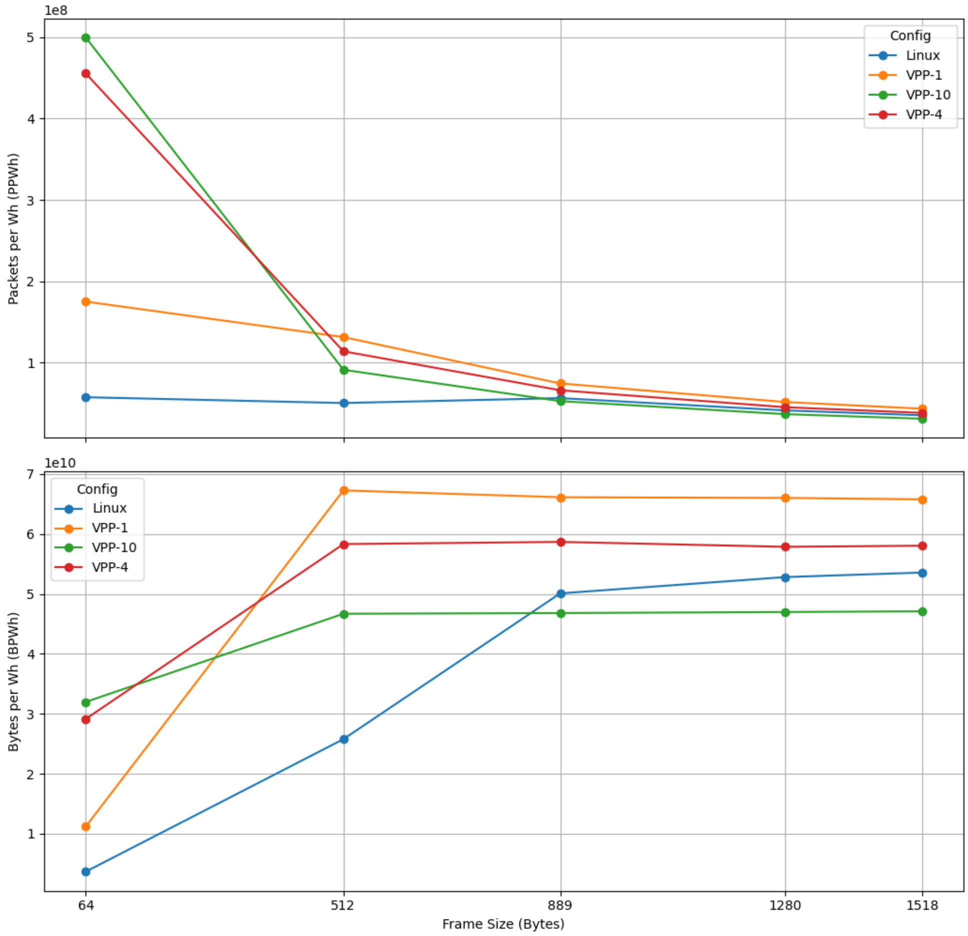

Figure 2.3 Energy efficiency per delivered data in one-way 25 Gbit/s.

As the results in Table 2.19 suggest, only the VPP-1 and Linux configurations were unable to deliver all data, while VPP-4 and VPP-10 showed significant improvements in latency. The average latency of the Linux stack increased, which may be attributed to a larger number of packets being successfully delivered.

Table 2.19 Results of one-way 40 Gbit/s of 512-byte frames

Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
VPP-1	5.82	35.26	292.45	122.15
VPP-4	6.56	0.00	28.80	9.90
VPP-10	8.00	0.00	35.45	14.35
Linux	TODO			

In the 889-byte frame test, the results for VPP-4 and VPP-10 remained consistent with the previous experiment. VPP-1 was almost able to handle the full traffic, while the Linux stack managed to process only about half of the data – and did so with extremely high latency. The results are summarized in Table 2.20.

Table 2.20 Results of one-way 40 Gbit/s of 889-byte frames

Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
VPP-1	5.77	3.85	203.05	25.2
VPP-4	6.54	0.00	32.55	13.75
VPP-10	8.00	0.00	33.60	18.50
Linux	7.63	49.85	7222.80	315.40

As the data in Table 2.21 suggest, all VPP configurations performed similarly to the corresponding 25 Gbit/s test, with the exception of VPP-1, which showed higher latency likely due to the increased traffic load. The Linux stack was almost able to process all data, showing a significant improvement compared to the 889-byte frame test.

Table 2.21 Results of one-way 40 Gbit/s of 1280-byte frames

Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
VPP-1	5.79	0.00	32.50	14.65
VPP-4	6.40	0.00	33.95	14.35
VPP-10	8.02	0.00	32.15	16.85
Linux	7.59	6.25	2830.95	104.1

The data in Table 2.22 demonstrate that the VPP configurations returned results comparable to the 1280-byte frame test, indicating that the architecture has already reached its efficiency limit under the tested conditions. In contrast, the Linux stack benefited from the increased frame size, being able to almost deliver the full traffic volume with significantly reduced latency.

Table 2.22 Results of one-way 40 Gbit/s of 1518-byte frames

Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
VPP-1	5.83	0.00	30.35	15.75
VPP-4	6.43	0.00	34.20	14.75
VPP-10	8.02	0.00	33.30	14.80
Linux	7.36	0.08	195.05	122.05

2.4 Presentation and Analysis of Results

Chapter 3
Conclusion

Appendix A
Nějaká příloha

Sem přijde to, co nepatří do hlavní části.

Bibliography

1. GALLATIN, Andrew J.; CHASE, Jeffrey S.; YOCUM, Kenneth G. Trapeze/IP: TCP/IP at Near-Gigabit Speeds. In: *Proceedings of the USENIX Annual Technical Conference*. 1999, pp. 109–120. Available also from: `https://www.usenix.org/event/usenix99/full_papers/gallatin/gallatin.pdf`.
2. COX, Alan L.; SCHAELICKE, Lambert; DAVIS, Al; MCKEE, Sally A. Profiling I/O Interrupts in Modern Architectures. *Proceedings of the Workshop on Performance Analysis and Its Impact on Design*. 2000. Available also from: `https://users.cs.utah.edu/~ald/pubs/interrupts.pdf`.
3. FD.IO. *What is VPP?* [`https://wiki.fd.io/view/VPP/What_is_VPP%3F`]. 2025. Accessed: 2025-04-07.
4. LINGUAGLOSSA, Leonardo; ROSSI, Dario; PONTARELLI, Salvatore; BARACH, Dave; MARJON, Damjan; PFISTER, Pierre. High-speed data plane and network functions virtualization by vectorizing packet processing. *Computer Networks*. 2019, vol. 149, pp. 187–199. ISSN 1389-1286. Available from DOI: `https://doi.org/10.1016/j.comnet.2018.11.033`.
5. BARACH, David; LINGUAGLOSSA, Leonardo; MARION, Damjan; PFISTER, Pierre; PONTARELLI, Salvatore; ROSSI, Dario. High-speed Software Data Plane via Vectorized Packet Processing. *IEEE Communication Magazine* [https://perso.telecom-paristech.fr/drossi/paper/rossi18commag.pdf]. 2018, vol. 56, no. 12, pp. 97–103. ISSN 0163-6804. Available from DOI: `10.1109/MCOM.2018.1800069`.
6. FD.IO. *Extensible: VPP and its plugin architecture* [`https://fd.io/docs/vpp/v2101/whatisvpp/extensible`]. 2021. Accessed: 2025-04-10.
7. FD.IO PROJECT. *Multi-threading in VPP*. 2025. Available also from: `https://s3-docs.fd.io/vpp/25.06/developer/corearchitecture/multi_thread.html`. Accessed: 2025-05-03.

8. DPDK PROJECT. *About DPDK* [https://www.dpdk.org/about/]. 2025. Accessed: 2025-04-13.
9. FREITAS, Eduardo; DE OLIVEIRA FILHO, Assis T.; DO CARMO, Pedro R.X.; SADOK, Djamel; KELNER, Judith. A survey on accelerating technologies for fast network packet processing in Linux environments. *Computer Communications*. 2022, vol. 196, pp. 148–166. ISSN 0140-3664. Available from DOI: https://doi.org/10.1016/j.comcom.2022.10.003.
10. DPDK PROJECT. *Supported NICs* [https://core.dpdk.org/supported/nics/]. 2024. Accessed: 2025-04-20.
11. BURAKOV, Anatoly. Memory in DPDK, Part 1: General Concepts. *DPDK Blog*. 2019. Available also from: https://www.dpdk.org/memory-in-dpdk-part-1-general-concepts/. Accessed: 2025-04-20.
12. ZHANG, Shiqi; GUPTA, Mridul; DEZFOULI, Behnam. *Understanding and Enhancing Linux Kernel-based Packet Switching on WiFi Access Points*. 2024. Available from arXiv: 2408.01013 [cs.NI].
13. DOCUMENTATION, Linux Kernel. *Scaling in the Linux Networking Stack* [https://docs.kernel.org/networking/scaling.html]. 2023. Accessed: 2025-04-19.
14. INTEL CORPORATION. *Core Utilization in DPDK Applications* [https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookbook/2025-0/core-utilization-in-dpdk-apps.html]. 2024. Accessed: 2025-04-19.
15. INTEL CORPORATION. *PCIe Traffic in DPDK Applications* [https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookbook/2025-0/pcie-traffic-in-dpdk-apps.html]. 2024. Accessed: 2025-04-19.
16. FD.IO PROJECT. *VPP Software Architecture*. 2025. Available also from: https://s3-docs.fd.io/vpp/25.06/developer/corearchitecture/softwarearchitecture.html. Accessed: 2025-04-21.
17. FD.IO PROJECT. *VPP Infrastructure Layer*. 2025. Available also from: https://s3-docs.fd.io/vpp/25.06/developer/corearchitecture/infrastructure.html. Accessed: 2025-04-21.
18. FD.IO PROJECT. *VNET - VPP Network Stack*. 2025. Available also from: https://s3-docs.fd.io/vpp/25.06/developer/corearchitecture/vnet.html. Accessed: 2025-04-21.
19. FD.IO PROJECT. *Feature Arcs*. 2025. Available also from: https://s3-docs.fd.io/vpp/25.06/developer/corearchitecture/featurearcs.html. Accessed: 2025-04-21.

20. *VLIB Architecture — VPP Developer Documentation (Release 25.06)*. FD.io Project, 2025. Available also from: `https://s3-docs.fd.io/vpp/25.06/developer/corearchitecture/vlib.html`. Accessed: 2025-04-21.
21. FD.IO PROJECT. *Plugins* [online]. 2025. [visited on 2025-04-21]. Available from: `https://s3-docs.fd.io/vpp/25.06/developer/plugins/index.html`.
22. FD.IO PROJECT. *Add a New Plugin* [online]. 2025. [visited on 2025-04-21]. Available from: `https://s3-docs.fd.io/vpp/25.06/developer/plugindoc/add_plugin.html`.
23. FD.IO PROJECT. *Supported Architectures and Operating Systems*. 2025. Available also from: `https://s3-docs.fd.io/vpp/25.06/aboutvpp/supported.html`. Accessed: 2025-05-03.
24. FD.IO PROJECT. *FD.io release package repository on Packagecloud.io*. 2025. Available also from: `https://packagecloud.io/fdio`. Accessed: 2025-05-03.
25. FD.IO PROJECT. *Building VPP*. 2025. Available also from: `https://s3-docs.fd.io/vpp/25.06/developer/build-run-debug/building.html`. Accessed: 2025-05-03.
26. FD.IO PROJECT. *Running VPP*. 2025. Available also from: `https://s3-docs.fd.io/vpp/25.06/gettingstarted/running/index.html`. Accessed: 2025-05-03.
27. DPDK PROJECT. *Linux Drivers*. 2025. Available also from: `https://doc.dpdk.org/guides/linux_gsg/linux_drivers.html`. Accessed: 2025-05-03.
28. FD.IO PROJECT. *Getting Started with the Configuration*. 2025. Available also from: `https://s3-docs.fd.io/vpp/25.06/configuration/config_getting_started.html`. Accessed: 2025-05-03.
29. FD.IO PROJECT. *Configuration Reference*. 2025. Available also from: `https://s3-docs.fd.io/vpp/25.06/configuration/reference.html`. Accessed: 2025-05-03.
30. FD.IO PROJECT. *VPP Feature List – FD.io Documentation (Release 25.02)*. 2025. Available also from: `https://s3-docs.fd.io/vpp/25.02/aboutvpp/featurelist.html`. Accessed: 2025-04-15.
31. FD.IO PROJECT. *FD.io VPP: High Performance, Modular, and Production Quality Software Forwarder*. 2017-07. Tech. rep. FD.io. Available also from: `https://fd.io/docs/whitepapers/FDioVPPwhitepaperJuly2017.pdf`. Accessed: 2025-04-16.

32. SLAVIC, Goran; KRAJNOVIC, Nenad. Practical implementation of the vector packet processing software router. In: *2024 32nd Telecommunications Forum (TELFOR)*. 2024, pp. 1–4. Available from DOI: 10.1109/TELFOR63250.2024.10819057.

33. ADELEKE, Oluwamayowa Ade; BASTIN, Nicholas; GURKAN, Deniz. Network Traffic Generation: A Survey and Methodology. *ACM Comput. Surv.* 2022, vol. 55, no. 2. ISSN 0360-0300. Available from DOI: 10.1145/3488375.

34. TEAM, The iPerf. *iPerf - The Network Testing Tool*. 2025. Available also from: https://iperf.fr/. Accessed: 2025-04-16.

35. COMPUTER NETWORKING GROUP, Università degli Studi di Napoli Federico II. *D-ITG: Distributed Internet Traffic Generator Manual*. 2013. Available also from: https://traffic.comics.unina.it/software/ITG/manual/. Accessed: 2025-04-16.

36. CISCO SYSTEMS, Inc. *TRex - Traffic Generator*. 2025. Available also from: https://trex-tgn.cisco.com/. Accessed: 2025-04-16.

37. PROJECT, DPDK. *Pktgen-DPDK* [https://pktgen-dpdk.readthedocs.io/]. 2025. Accessed: 2025-04-16.

38. *Benchmarking Methodology for Network Interconnect Devices* [RFC 2544]. RFC Editor, 1999. Request for Comments, no. 2544. Available from DOI: 10.17487/RFC2544.

39. ALLAN, David; MARTINEZ, Jordi Palet. *Benchmarking Methodology for IPv6 Transition Technologies* [https://datatracker.ietf.org/doc/html/rfc8219]. 2017. RFC 8219.

40. JURKIEWICZ, Piotr; RZYM, Grzegorz; BORYŁO, Piotr. Flow length and size distributions in campus Internet traffic. *Computer Communications*. 2021, vol. 167, pp. 15–30. ISSN 0140-3664. Available from DOI: https://doi.org/10.1016/j.comcom.2020.12.016.

Obsah příloh

```
/
├── readme.txt ............................ stručný popis obsahu média
├── exe ..................... adresář se spustitelnou formou implementace
├── src
│   ├── impl ................................ zdrojové kódy implementace
│   └── thesis ................... zdrojová forma práce ve formátu LaTeX
└── text .................................................... text práce
    └── thesis.pdf ......................... text práce ve formátu PDF
```