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 16, 2025

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Slavík** Jméno: **Ondřej** Osobní číslo: **516039**

Fakulta/ústav: **Fakulta informačních technologií**

Zadávající katedra/ústav: **Katedra počítačových systémů**

Studijní program: **Informatika**

Specializace: **Počítačové sítě a Internet**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Efektivita a využití Vector Packet Processing ve vysokorychlostních sítích

Název bakalářské práce anglicky:

Efficiency and utilization of Vector Packet Processing in high-speed networks

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Jan Fesl, Ph.D. katedra počítačových systémů FIT

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **29.01.2025** Termín odevzdání bakalářské práce: ___________

podpis vedoucí(ho) ústavu/katedry

podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

Datum převzetí zadání

Slavík Ondřej

Podpis studenta

ČVUT ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Slavík** Jméno: **Ondřej** Osobní číslo: **516039**

Fakulta/ústav: **Fakulta informačních technologií**

Zadávající katedra/ústav: **Katedra počítačových systémů**

Studijní program: **Informatika**

Specializace: **Počítačové sítě a Internet**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Efektivita a využití Vector Packet Processing ve vysokorychlostních sítích

Název bakalářské práce anglicky:

Efficiency and utilization of Vector Packet Processing in high-speed networks

Pokyny pro vypracování:

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.

Seznam doporučené literatury:

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.

I declare that I have used AI tools during the preparation and writing of my thesis. I have verified the generated content. I confirm that I am aware that I am fully responsible for the content of the thesis.

In Prague on May 16, 2025

Abstract

This bachelor's thesis focuses on the efficiency and utilization of the Vector Packet Processing (VPP) framework in high-speed networks. The aim is to experimentally compare its packet processing performance with the traditional kernel-based networking stack. The thesis first introduces the theoretical foundations of VPP and its architecture. In the practical part, test scenarios are designed and executed to evaluate and compare both solutions. The results show that VPP achieves higher throughput and lower latency at higher transmission speeds, with energy efficiency varying depending on traffic volume and configuration.

Keywords vector packet processing, data plane development kit, network benchmark, energy efficiency, Linux network stack

Abstrakt

Tato bakalářská práce se zabývá efektivitou a využitím frameworku Vector Packet Processing (VPP) ve vysokorychlostních sítích. Cílem je experimentálně porovnat jeho výkon při zpracování síťových paketů s tradičním síťovým stackem v jádře operačního systému. Práce nejprve představuje teoretické základy VPP a popisuje jeho architekturu. V praktické části jsou navrženy a realizovány testovací scénáře pro porovnání obou řešení. Výsledky ukazují, že VPP dosahuje vyšší propustnosti a nižší latence při vyšších přenosových rychlostech, přičemž energetická efektivita se liší v závislosti na objemu provozu a použité konfiguraci.

Klíčová slova vector packet processing, data plane development kit, síťové měření, energetická efektivita, síťový stack Linuxu

Contents

List of Figures

List of Tables

List of code listings

List of abbreviations

API	Application Programming Interface
BIRD	BIRD Internet Routing Daemon
BPWh	Bytes Per Watt-Hour
CLI	Command-Line Interface
CNF	Cloud-Native Network Function
DMA	Direct Memory Access
DPDK	Data Plane Development Kit
DUT	Device Under Test
ECMP	Equal-Cost Multi-Path
FD.io	Fast Data Project
GRO	Generic Receive Offload
GRE	Generic Routing Encapsulation
GTP-U	GPRS Tunneling Protocol - User plane
IRQ	Interrupt Request
L2	Layer 2 (Data Link Layer)
L3	Layer 3 (Network Layer)
L4	Layer 4 (Transport Layer)
Linux	Linux kernel network stack
LTS	Long Term Support
MTU	Maximum Transmission Unit
NAT	Network Address Translation
NFV	Network Function Virtualization
NIC	Network Interface Card
NUMA	Non-Uniform Memory Access
PMD	Poll Mode Driver
PPWh	Packets Per Watt-Hour
RSS	Receive Side Scaling
SDN	Software Defined Networking
VLIB	Vector Library
VNET	Vector Network Layer
VNF	Virtual Network Function
VPP	Vector Packet Processing
VXLAN	Virtual Extensible LAN

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 focus on comparing throughput, latency, and energy efficiency across different VPP configurations and a reference Linux-based router under various traffic conditions. The goal is to understand the theoretical advantages of VPP, design and implement a suitable measurement infrastructure, define representative traffic scenarios, and ultimately evaluate whether VPP provides superior performance. The aim is also to characterize the specific conditions and configurations under which it outperforms the Linux-based reference system. The

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

results may support a recommendation for its deployment by small-to-mid-sized ISPs currently using Linux-based routers or planning to upgrade legacy systems. Moreover, in light of the increasing focus on energy efficiency in IT infrastructure, the results may also inform ongoing discussions about sustainable network architecture.

With the development of AI and the growing demand for high-resolution streaming services, it is highly likely that the demand for internet bandwidth 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. 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 instruction caches. [2]

1.1.2 An Introduction to VPP

Vector Packet Processing 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.

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 gains. 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 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 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]

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.

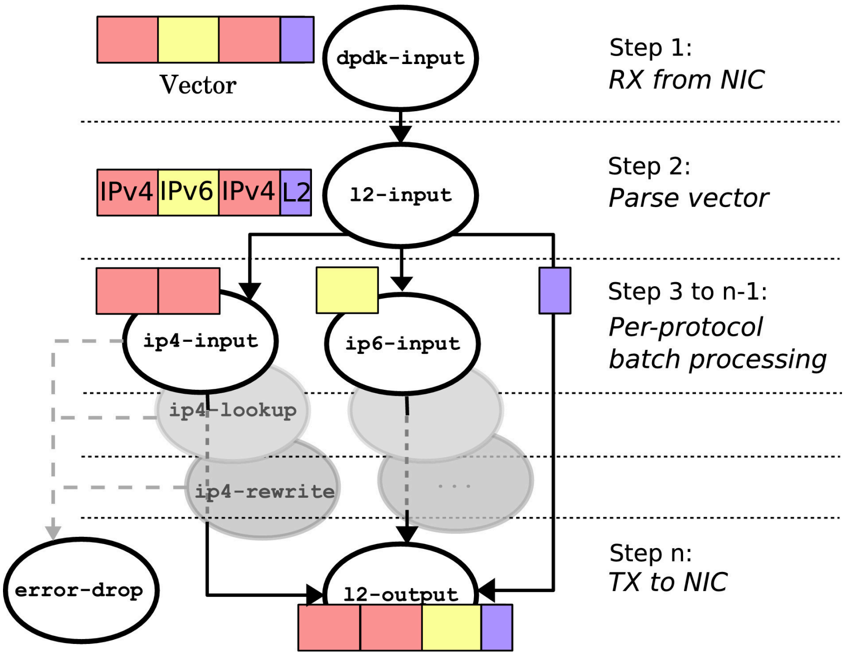

Figure 1.1 Picture showing the VPP Processing Graph [4]

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 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 – 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, 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 interface card (NIC) using direct memory access (DMA), which writes packet data into pre-allocated memory buffers specified by receive (Rx) descriptors.

[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.

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]

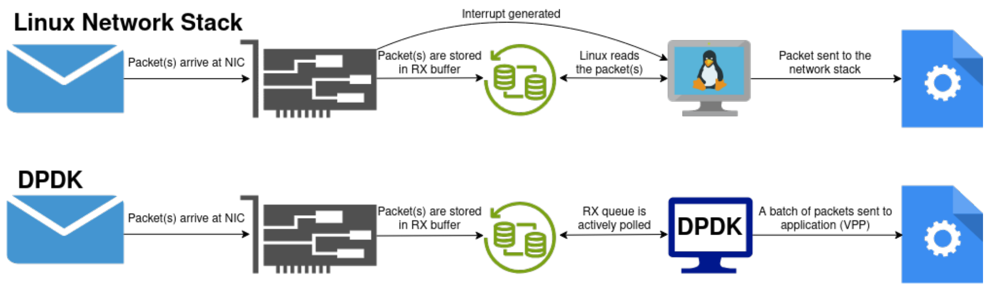

Figure 1.2 Diagram illustrating the differences in packet handling between the Linux Network Stack and DPDK [12, 13, 14, 15]

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. 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.

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

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 pro-

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

viding 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 { interactive cli-listen 127.0.0.1:5002 }`, 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]

- **The buffers section** allows configuring the size and number of memory buffers used by VPP for packet processing. [29]

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

- **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 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 TRex in the subsequent experimental evaluation. TRex was chosen for its modern architecture, support for high-speed stateful and stateless traffic generation, and ability to simulate real-world traffic patterns. 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 were excluded due to either their complex usage (Pktgen-DPDK), limited maintenance (D-ITG), or limited feature set (iPerf3).

Chapter 2

Practical part

2.1 Building Infrastructure for Measurement

The testing infrastructure has been implemented as recommended 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].

The Device Under Test (DUT) and the measurement device are connected using 100 Gbit capable cables, preventing any potential bottlenecks in the connection. The illustration of this hardware setup is shown in Fig. 2.1

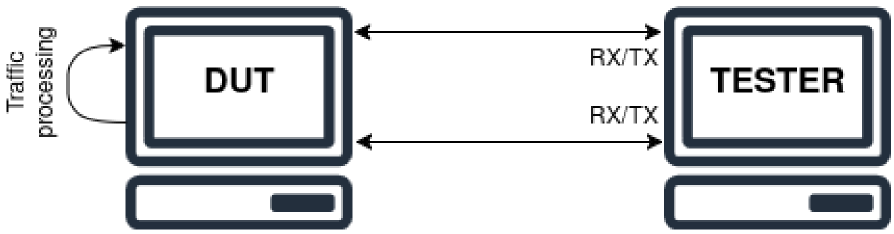

Figure 2.1 Picture showing hardware setup

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. The DUT is responsible for processing network traffic and responding to the test conditions set by the measurement 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.

The Tester (Measurement Device), on the other hand, is responsible for generating the network traffic and capturing the responses from the DUT. Its

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

physical features are shown in Table 2.2.

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 (1 thread/core)
Memory (RAM)	256 GB of DDR4 ECC
NIC	Mellanox ConnectX-6 Dx (Dual-port)

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 (2 threads/core)
Memory (RAM)	384 GB DDR4 ECC
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. This kernel version is the current standard long-term support (LTS) release provided with Debian 12 (Bookworm) and was used for all tests involving the Linux networking stack. The tester generates traffic using Cisco TRex version 3.06.

2.2 Methodology

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 repeated tests were documented and clearly marked in the provided data.

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. It should also be noted that in cases where packet loss occurred, higher deviations in delay and jitter statistics can be expected, as packet loss may negatively impact the consistency of these measurements. 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 and 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. [39], 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. These transmission rates reflect standard Ethernet link speeds as defined in IEEE 802.3 [40].

All traffic was generated using TRex with the profile included in Appendix A, which ensures that each packet carries a varying source IP address to simulate multiple concurrent clients, while maintaining a single destination IP per direction. In all forwarding scenarios, a /24 source IP pool was used, while a /16 pool was employed in NAT scenarios to simulate a higher number of concurrent clients. 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. In the tables, the number of worker threads is denoted as VPP-X, where X indicates the number of worker threads used.

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. Affinities are 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. All CPU cores used for testing (including VPP scenarios) are set to performance mode, as is standard practice in network workloads.

In order to cover common real-world deployment scenarios, three traffic patterns were evaluated: one-way forwarding, bi-directional (both-way) forwarding, and NAT with forwarding. These scenarios represent typical use cases encountered in small to medium-sized ISP or enterprise environments, where Linux or VPP may be deployed as the primary software-based router. NAT is a commonly used feature in smaller ISP deployments. As it is a computationally intensive operation, the key question in this context is how it affects overall performance.

2.3.1 One-way forwarding

These tests were conducted in a one-way configuration, as suggested by RFC 2544 [38]. Such setup can represent asymmetric traffic patterns commonly found in real-world scenarios – for example, downstream-heavy services like web servers or video streaming platforms. It can also emulate high-load conditions similar to those observed during denial-of-service (DoS) attacks, where a large volume of traffic targets a single destination.

2.3.1.1 1 Gbps Test Results

As can be seen from the results presented in Table 2.3, all configurations were able to successfully transmit all data in this low-traffic scenario. The VPP-1 configuration performed best across all metrics. Even though this was the least demanding test scenario, the Linux stack still consumed more power than VPP-1. In terms of latency, the Linux configuration outperformed both VPP-4 and VPP-10 at frame sizes of 1280B and 1518B. However, latency in the Linux setup increased when switching from 1280B to 1518B frames, which can likely be attributed to internal kernel memory copying and buffer processing. In contrast, all VPP configurations either maintained or slightly reduced latency with increasing frame size, demonstrating better scalability. Interestingly, the single-threaded VPP-1 configuration achieved the lowest latency among all VPP variants. This can likely be attributed to the absence of L3 cache thrashing and the ability to process more packets in a single batch, which amortizes the per-packet processing cost – unlike parallelized VPP, which must traverse the processing graph with fewer packets per vector. In low-demand scenarios, multithreading may become a disadvantage rather than an advantage.

The energy efficiency of all configurations is shown in Fig. 2.2. All VPP configurations maintained stable BPWh values, which is due to their busy-wait processing model. The Linux stack becomes more efficient with increasing

Table 2.3 Results of one-way 1 Gbit/s tests

	Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
64B	VPP-1	5.68	0.00	12.80	9.50
	VPP-4	6.46	0.00	27.45	13.55
	VPP-10	7.86	0.00	28.30	12.65
	Linux	6.78	0.00	108.05	97.25
512B	VPP-1	5.69	0.00	12.10	11.70
	VPP-4	6.48	0.00	22.85	17.40
	VPP-10	7.91	0.00	23.80	17.35
	Linux	6.23	0.00	56.10	51.35
889B	VPP-1	5.76	0.00	10.30	8.40
	VPP-4	6.45	0.00	21.30	17.60
	VPP-10	7.85	0.00	20.95	16.30
	Linux	6.14	0.00	23.70	21.50
1280B	VPP-1	5.72	0.00	8.10	5.80
	VPP-4	6.46	0.00	18.30	17.20
	VPP-10	7.84	0.00	18.80	15.95
	Linux	6.11	0.00	9.95	3.80
1518B	VPP-1	5.66	0.00	7.20	5.50
	VPP-4	6.42	0.00	16.10	15.75
	VPP-10	7.84	0.00	17.65	17.35
	Linux	6.10	0.00	12.25	10.55

frame size, likely as a result of less frequent system calls. As can be seen, even in the most favourable scenario, the Linux stack was not more energy-efficient than VPP-1, and when transmitting 64-byte frames, it was also less efficient than VPP-4.

2.3.1.2 10 Gbps Test Results

In the 10 Gbit/s scenario, both VPP-1 and the Linux stack experienced significant difficulties delivering 64-byte frames, resulting in high packet loss and dramatically increased latency. In contrast, all other configurations handled the traffic without loss, and latency generally decreased with increasing frame size. A minor deviation from this trend was observed in the VPP-1 512-byte and VPP-4 64-byte frame tests, where latency temporarily increased with larger frame sizes before dropping again. This suggests that VPP configurations may deliver optimal latency when operating near their forwarding capacity. As observed before, VPP-1 consistently achieved the lowest latency among all VPP variants, except in the 64-byte test. The Linux stack once again exhibited a latency increase when transitioning from 1280B to 1518B frames, as previously observed in the 1Gbit/s tests. The detailed statistics are shown in Tab. 2.4

Figure 2.3 illustrates the energy efficiency of each configuration in this test.

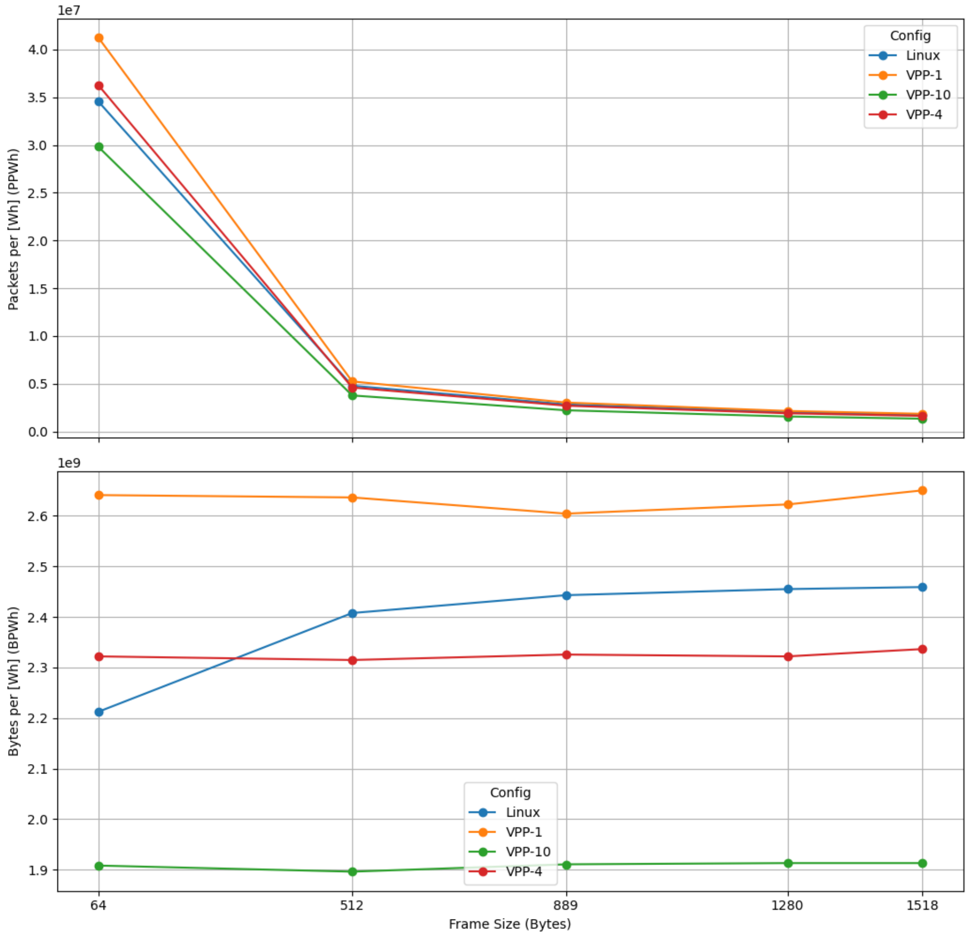

Figure 2.2 Energy efficiency per delivered data in one-way 1 Gbit/s

Table 2.4 Results of one-way 10 Gbit/s tests

	Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
64B	VPP-1	5.72	59.50	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	217.80
512B	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	129.40	99.35
889B	VPP-1	5.62	0.00	23.40	14.95
	VPP-4	6.48	0.00	26.50	18.25
	VPP-10	7.95	0.00	26.95	17.55
	Linux	6.67	0.00	66.45	68.30
1280B	VPP-1	5.58	0.00	22.80	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
1518B	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

The significant drop in performance for VPP-1 and Linux during the 64-byte frame test is attributed to high packet loss. When all packets are successfully delivered, all VPP configurations maintain stable BPWh values, as observed in the previous test. In contrast, the Linux stack showed a decline in performance across all frame sizes.

2.3.1.3 25 Gbps Test Results

As shown in Table 2.5, none of the configurations were able to deliver all 64-byte frames, which naturally resulted in high latency in that test. It can be observed that increasing the frame size beyond 889 bytes had no measurable impact on the performance of any VPP configuration – the results remain comparable to the previous test. In contrast, the Linux stack benefited from larger frames, and the increase in latency in prevoius test seen in the 1518-byte scenario compared to the 1280-byte scenario did not occur at this higher transmission rate. An interesting observation is that when VPP drops packets, the ones that are not dropped tend to be delivered with significantly lower latency.

Figure 2.4 illustrates the energy efficiency observed during the 25 Gbit/s test. The drop in BPWh across all configurations in 64-byte frame test is

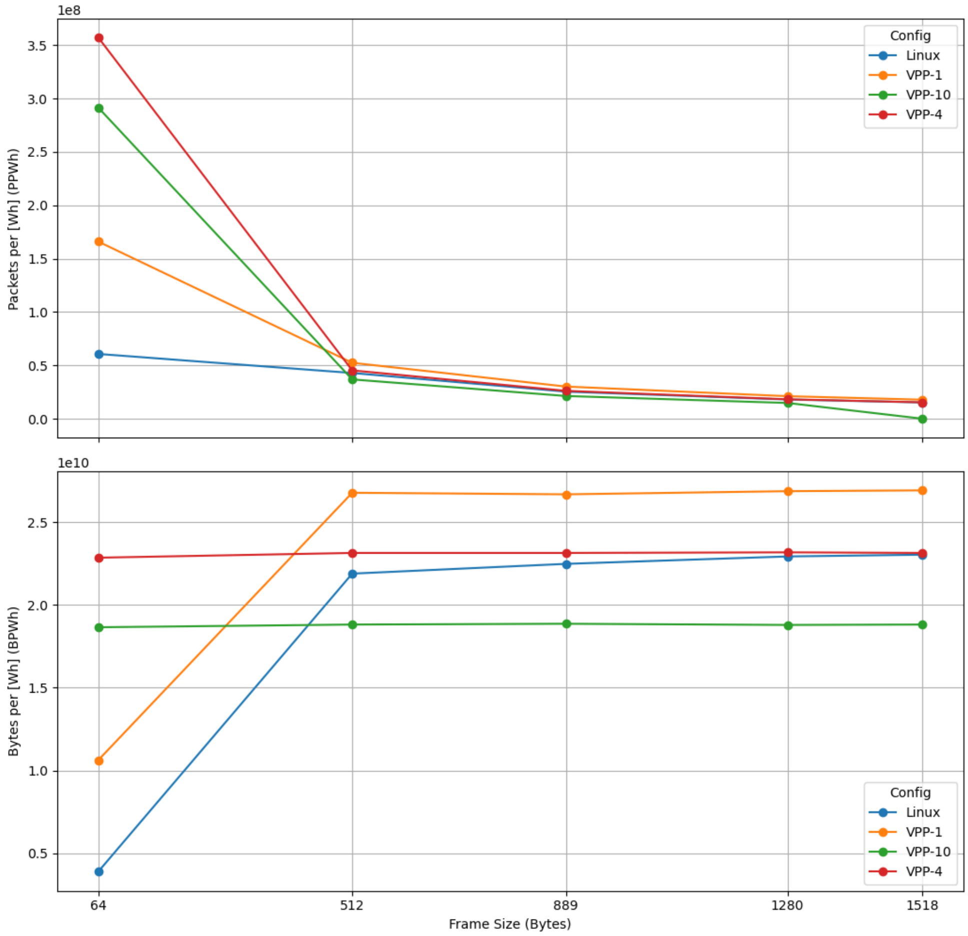

Figure 2.3 Energy efficiency per delivered data in one-way 10 Gbit/s

Table 2.5 Results of one-way 25 Gbit/s tests

	Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
64B	VPP-1	5.59	83.29	577.50	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
512B	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
889B	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
1280B	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
1518B	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	28.90	15.65
	Linux	7.00	0.00	130.00	105.30

primarily caused by high packet loss. Compared to the previous test, the Linux stack performed significantly worse – even at the largest frame size – while all VPP configurations maintained stable BPWh values, except in the case of 64-byte frames.

2.3.1.4 40 Gbps Test Results

As shown in Table 2.6, the 64-byte frame test resulted in even higher packet loss than in the previous test. However, the latency statistics for VPP remained comparable to those observed earlier, despite the increased loss. In this test, the Linux stack failed to deliver all data across all frame sizes, which again led to consistently high latency. When no data were dropped, all VPP configurations performed similarly, maintaining low and stable latency. These results clearly highlight the advantage of VPP over the Linux network stack under heavy traffic loads. Notably, even the least resource-intensive VPP configuration (VPP-1) significantly outperformed the Linux stack in both latency and reliability, demonstrating VPP's efficiency even with minimal parallelism.

The energy efficiency graph shown in Fig. 2.5 illustrates that VPP maintained stable BPWh values when no packets were dropped, while the Linux

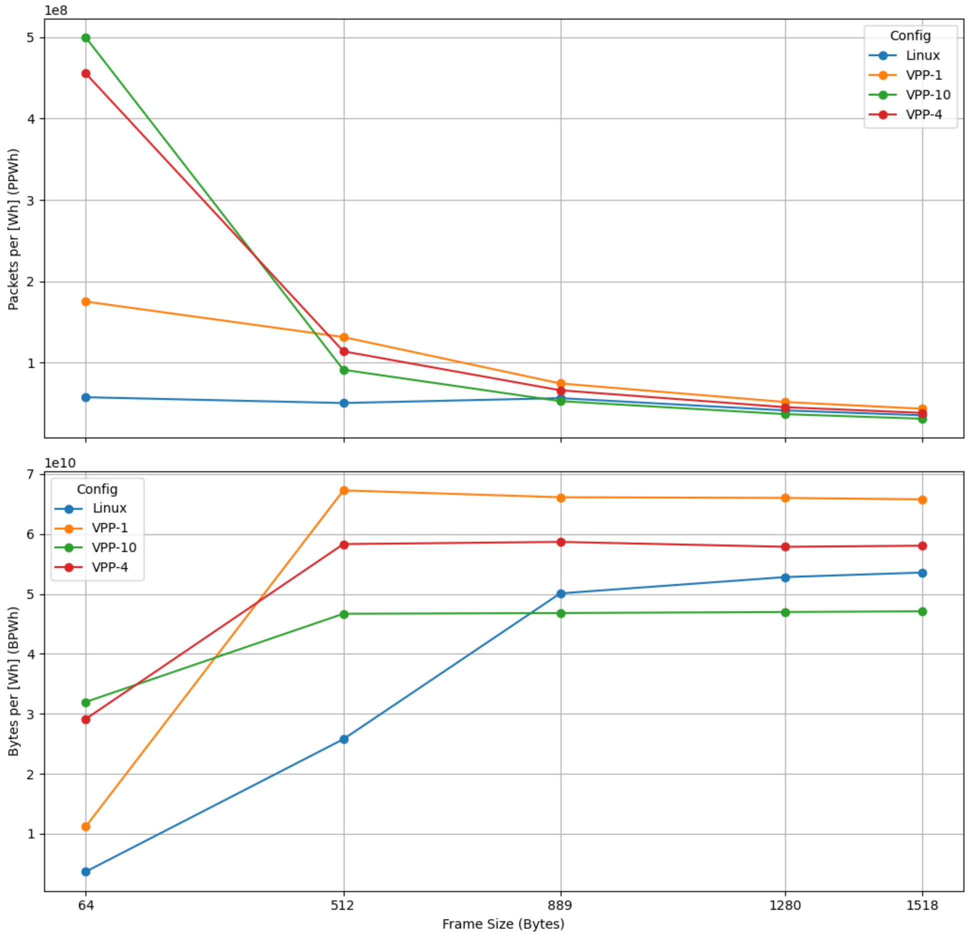

Figure 2.4 Energy efficiency per delivered data in one-way 25 Gbit/s

Table 2.6 Results of one-way 40 Gbit/s tests

	Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
64B	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
512B	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	7.56	69.84	6621.50	892.35
889B	VPP-1	5.77	3.85	203.05	25.20
	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
1280B	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	104.10
1518B	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

stack again exhibited even lower energy efficiency per delivered data. This result confirms and completes the trend observed in the previous tests.

2.3.2 Bidirectional forwarding

In accordance with the recommendations of RFC 8219 [41], which – although focused on IPv6 transition technologies – also defines general benchmarking principles, the same test scenarios were executed under bidirectional traffic conditions. This configuration reflects more realistic networking environment where traffic flows simultaneously in both directions, such as in point-to-point communications, VPN tunnels, or client-server interactions involving both requests and responses. Unlike the one-way setup, bidirectional forwarding places a greater strain on system resources by utilizing both receive (RX) and transmit (TX) paths concurrently, which may reveal bottlenecks or limitations not evident in unidirectional traffic scenarios. It is important to note that bidirectional tests generate traffic at a rate of X Gbit/s in each direction, resulting in a total system load of $2X$ Gbit/s.

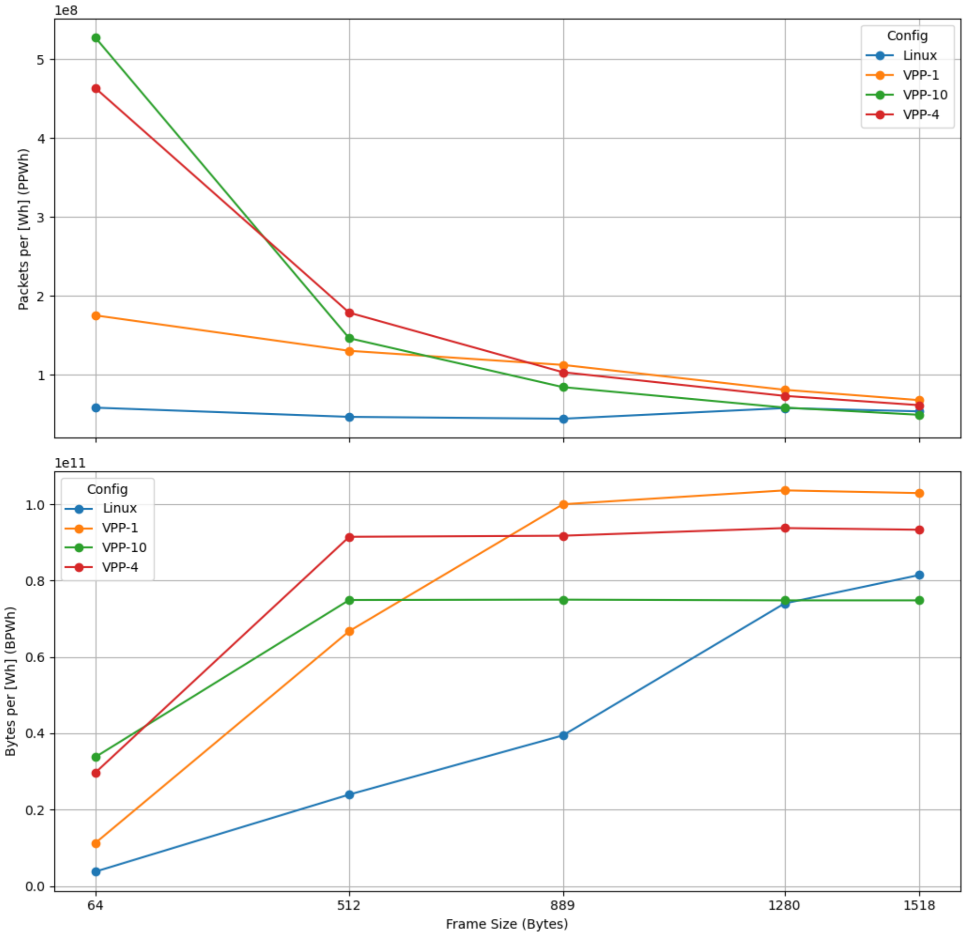

Figure 2.5 Energy efficiency per delivered data in one-way 40 Gbit/s

2.3.2.1 1 Gbps Test Results

As shown in Table 2.7, bidirectional forwarding at 1 Gbit/s resulted in increased latency across all VPP configurations compared to unidirectional traffic. VPP-4 and VPP-10 achieved latency levels similar to those observed in the 10 Gbit/s unidirectional test, except for the 64-byte frame case, which remained closer to the 1 Gbit/s results. Notably, VPP-1 performed significantly worse than in the unidirectional scenario, showing latency figures comparable to those seen in the 25 Gbit/s unidirectional test. This indicates that polling multiple interfaces negatively impacts latency performance in VPP. While VPP also polled both interfaces in the unidirectional tests, the secondary interface had no traffic, and was merely checked periodically. In the bidirectional case, however, VPP had to process packet vectors from both NICs, which clearly introduced additional latency. Since VPP processes packets in vectors from one interface at a time, traffic arriving on the second interface must wait until the current batch is completed. This serial processing model contributes to increased latency under bidirectional load. Interestingly, the Linux network stack outperformed all VPP configurations when average or large frames were used, achieving the lowest latency in these scenarios. By comparison, VPP delivered better results in the small frame tests, showcasing its capability to handle high-packet-rate workloads efficiently.

Figure 2.6 shows that the results for PPWh and BPWh are similar to those of the one-way 1 Gbit/s test, although Linux exhibits slightly lower efficiency compared to VPP then was observed in one-way scenario.

2.3.2.2 10 Gbps Test Results

The results of the 10 Gbit/s bidirectional test, presented in Table 2.8, are more comparable to the 25 Gbit/s one-way test than to the 10 Gbit/s one-way scenario, indicating that serving an additional NIC had no negative impact on performance. The only notable difference is that the VPP-1 configuration achieved better latency in one-way 25 Gbit/s forwarding for medium and large frames, despite handling a higher total traffic load. In the bidirectional test, with a total throughput of 20 Gbit/s, all VPP configurations – except in the 64-byte frame scenario – were able to handle the traffic successfully while maintaining stable latency. By contrast, the Linux stack struggled to process the full traffic load. Even in cases where it managed to deliver all packets (i.e., with larger frames), it still exhibited significantly higher latency.

The energy efficiency, in terms of BPWh and PPWh, is similar to that observed in the one-way 10 Gbit/s and 25 Gbit/s tests, with results being more closely aligned with the 25 Gbit/s case. Given the combined throughput of 20 Gbit/s, this behavior was expected. The corresponding results are visualized in Figure 2.7.

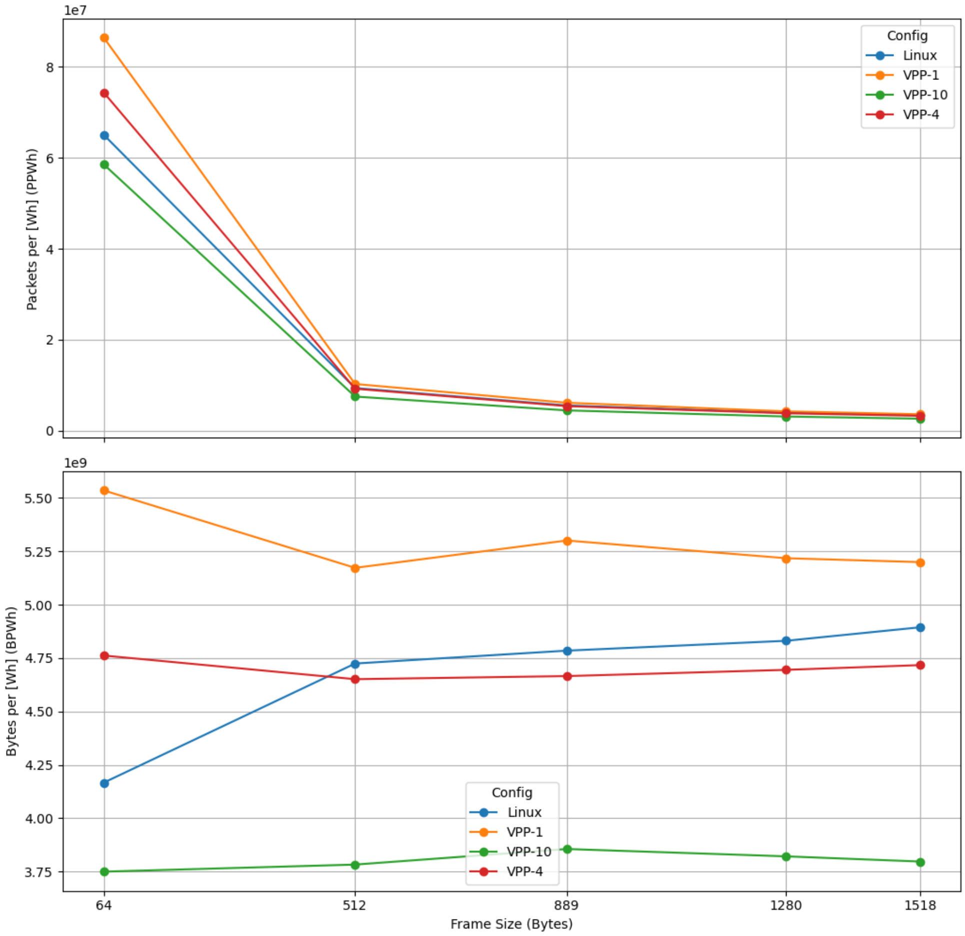

Figure 2.6 Energy efficiency per delivered data in bidirectional 1 Gbit/s

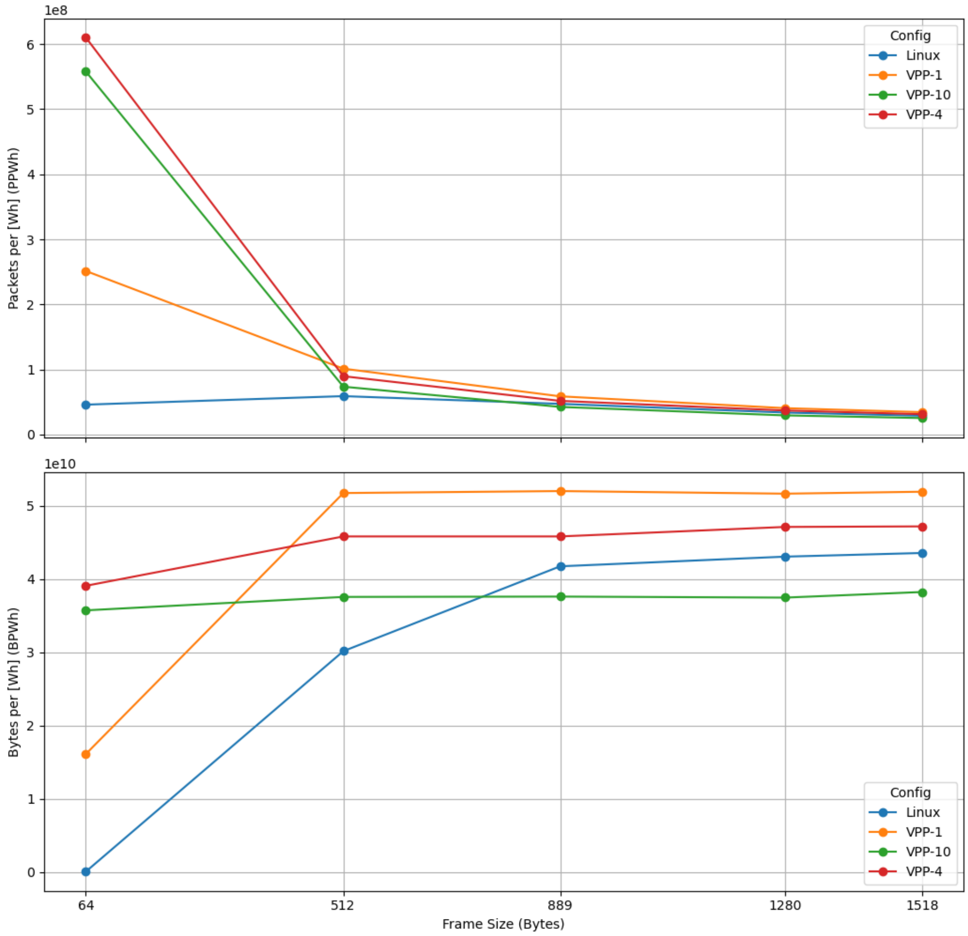

Figure 2.7 Energy efficiency per delivered data in bidirectional 10 Gbit/s

Table 2.7 Results of bidirectional 1 Gbit/s tests

	Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
64B	VPP-1	5.42	0.00	27.83	10.48
	VPP-4	6.30	0.00	30.18	14.08
	VPP-10	8.00	0.00	30.45	13.48
	Linux	7.20	0.00	147.68	100.05
512B	VPP-1	5.80	0.00	28.05	15.70
	VPP-4	6.45	0.00	26.73	17.30
	VPP-10	7.93	0.00	24.56	16.30
	Linux	6.35	0.00	39.83	39.20
889B	VPP-1	5.66	0.00	26.10	14.50
	VPP-4	6.43	0.00	26.13	17.53
	VPP-10	7.78	0.00	26.28	17.00
	Linux	6.27	0.00	17.4	12.38
1280B	VPP-1	5.75	0.00	25.20	12.45
	VPP-4	6.39	0.00	25.23	17.10
	VPP-10	7.85	0.00	27.40	18.33
	Linux	6.21	0.00	12.68	7.28
1518B	VPP-1	5.77	0.00	25.10	13.15
	VPP-4	6.36	0.00	25.53	15.63
	VPP-10	7.90	0.00	20.40	13.05
	Linux	6.13	0.00	12.40	9.40

2.3.2.3 25 Gbps Test Results

As with the previous test, the results presented in Table 2.9 more closely resemble those of the one-way 40 Gbit/s scenario. Since Linux was unable to deliver all packets in that test, it is not surprising that even more packets were dropped in this case. Interestingly, in all VPP configurations where packet loss occurred, the percentage of dropped packets was slightly lower compared to the one-way 40 Gbit/s test, despite the higher total throughput of 50 Gbit/s. One possible explanation is that bidirectional forwarding leads to more balanced utilization of RX queues across multiple cores. Moreover, in scenarios without packet loss, VPP maintained stable latency values – although increased compared to the previous 10 Gbit/s bidirectional experiment.

Concerning energy efficiency, it is worth noting that the Linux network stack remained the least energy-efficient across all frame sizes, primarily due to the high packet loss. The results closely resemble those observed in the one-way 40 Gbit/s test. A graph visualizing these results is shown in Figure 2.8.

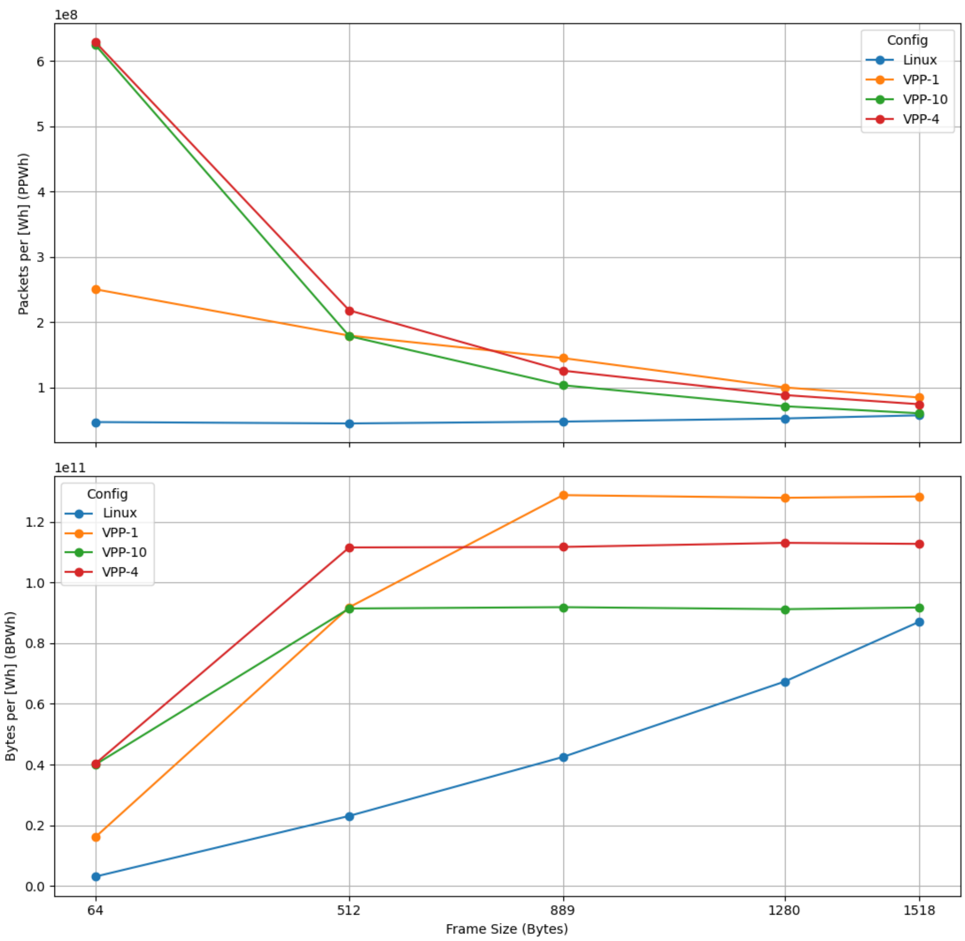

Figure 2.8 Energy efficiency per delivered data in bidirectional 25 Gbit/s

Table 2.8 Results of bidirectional 10 Gbit/s tests

	Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
64B	VPP-1	5.52	70.40	537.43	12.15
	VPP-4	6.51	30.47	188.40	7.23
	VPP-10	8.40	0.00	42.05	12.55
	Linux	7.24	92.91	8408.15	924.05
512B	VPP-1	5.80	0.00	27.75	11.73
	VPP-4	6.55	0.00	32.50	14.48
	VPP-10	7.99	0.00	30.58	15.38
	Linux	7.44	25.22	3837.13	172.03
889B	VPP-1	5.77	0.00	34.00	15.15
	VPP-4	6.55	0.00	32.50	14.48
	VPP-10	7.98	0.00	30.50	16.35
	Linux	7.19	0.00	110.20	82
1280B	VPP-1	5.81	0.00	34.15	14.86
	VPP-4	6.37	0.00	32.73	16.88
	VPP-10	8.01	0.00	32.20	15.85
	Linux	6.97	0.00	77.45	70.33
1518B	VPP-1	5.78	0.00	33.80	16.10
	VPP-4	6.36	0.00	25.56	15.63
	VPP-10	7.85	0.00	27.40	18.33
	Linux	6.89	0.00	72.78	70.08

2.3.2.4 40 Gbps Test Results

This final and most demanding test, with a total bidirectional throughput of 80 Gbit/s, clearly demonstrates the scalability of VPP compared to the Linux network stack. In every frame size scenario, Linux dropped more than half of all packets, confirming its limited capability to handle high-throughput workloads. VPP, on the other hand, managed to process a substantial portion of the traffic even in the smallest frame sizes, and at higher configurations (VPP-4 and VPP-10) was able to eliminate packet loss entirely for medium and large frames. These results further underline the advantages of user-space packet processing under extreme conditions. The results are summarised in Table 2.10.

As shown in the energy efficiency graph in Figure 2.9, Linux clearly remains the least energy-efficient configuration. Due to high packet loss, the energy efficiency of VPP-1 also declined, falling below that of VPP-4.

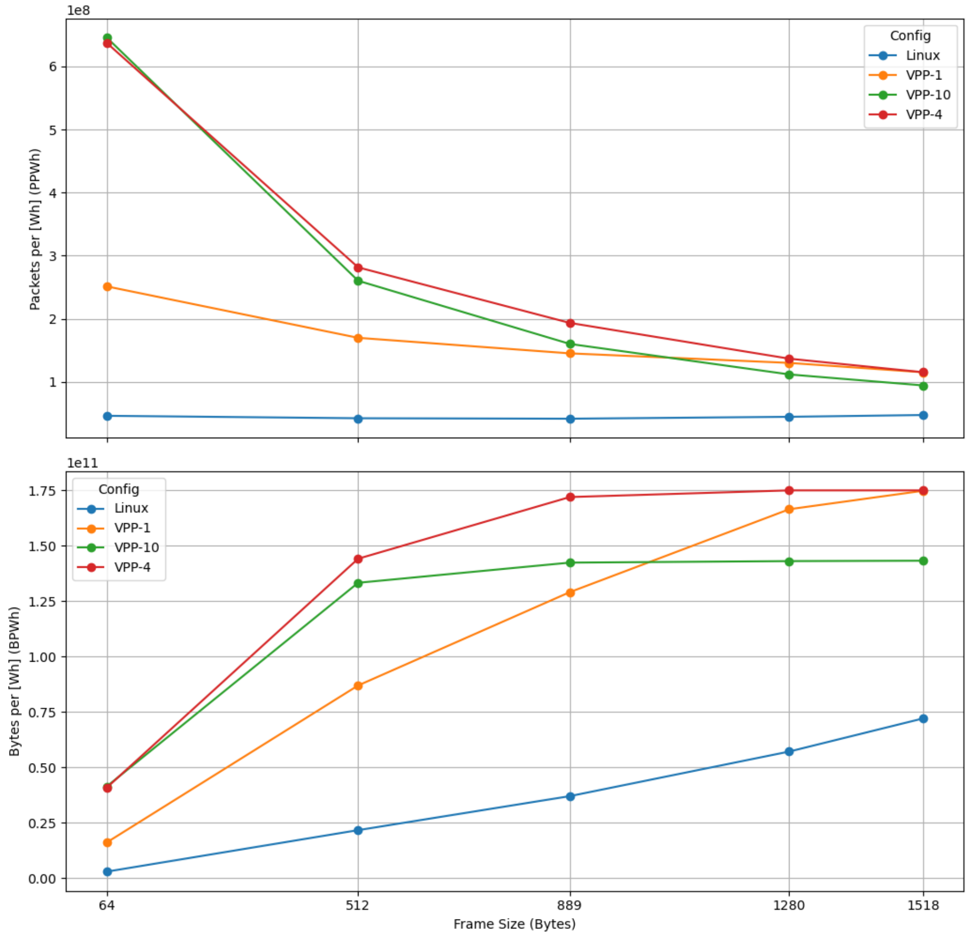

Figure 2.9 Energy efficiency per delivered data in bidirectional 40 Gbit/s

Table 2.9 Results of bidirectional 25 Gbit/s tests

	Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
64B	VPP-1	5.54	88.16	545.50	12.30
	VPP-4	6.60	64.58	185.08	8.18
	VPP-10	8.40	42.05	159.55	12.55
	Linux	7.23	97.09	9266.88	1409.90
512B	VPP-1	5.80	28.95	294.13	25.80
	VPP-4	6.72	0.00	38.18	13.28
	VPP-10	8.20	0.00	37.38	15.43
	Linux	7.53	76.86	8079.28	644.00
889B	VPP-1	5.82	0.01	43.63	17.15
	VPP-4	6.71	0.00	39.58	16.38
	VPP-10	8.16	0.00	36.83	16.65
	Linux	7.50	57.50	16506.75	814.28
1280B	VPP-1	5.86	0.00	40.40	15.73
	VPP-4	6.63	0.00	39.43	15.55
	VPP-10	8.22	0.00	37.93	15.25
	Linux	7.50	32.67	13791.63	357.95
1518B	VPP-1	5.84	0.00	43.95	18.58
	VPP-4	6.65	0.00	38.38	15.88
	VPP-10	8.17	0.00	36.28	17.78
	Linux	7.70	10.57	6568.30	112.68

2.3.3 NAT

Network Address Translation (NAT) is a commonly used technique among small to mid-sized ISPs for conserving public IP addresses and managing internal networks. Although NAT is a standard practice, it introduces additional complexity into packet forwarding. Specifically, NAT requires the router to maintain stateful information for each active connection. For every translated flow, the router must store the original and translated source IP addresses and ports, along with the destination address and protocol type. This information is necessary to correctly map incoming response packets back to the appropriate internal hosts. Since NAT modifies headers – such as the source IP addresses and source ports – all affected checksums must be recalculated, which imposes further computational overhead. The goal of this test is to evaluate how NAT impacts overall system performance.

For this purpose, source NAT is applied on top of the one-way forwarding scenario, meaning that the test setup is identical to the basic forwarding case, but with NAT translation enabled at the DUT. A /16 source IP address pool is used to emulate a large number of concurrent clients, while the DUT has a /24 pool of inside global IP addresses available for translation. This configuration serves as a stress test for the NAT implementation and its capacity to manage

Table 2.10 Results of bidirectional 40 Gbit/s tests

	Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
64B	VPP-1	5.52	92.60	543.45	13.25
	VPP-4	6.64	77.42	182.70	6.58
	VPP-10	8.38	71.13	156.08	10.375
	Linux	7.22	98.22	9256.2	1353.3
512B	VPP-1	5.82	57.83	317.70	103.95
	VPP-4	6.74	19.06	175.25	21.63
	VPP-10	8.42	13.00	168.70	20.28
	Linux	7.59	86.28	8976.75	924.63
889B	VPP-1	5.84	37.21	215.45	47.30
	VPP-4	6.89	1.29	142.7	17.2
	VPP-10	8.43	0.00	52.48	15.38
	Linux	7.64	76.43	17567.73	1555.53
1280B	VPP-1	5.89	18.32	179.13	36.65
	VPP-4	6.86	0.00	54.68	18.28
	VPP-10	8.39	0.00	49.63	17.40
	Linux	7.48	64.37	16067.48	673.3
1518B	VPP-1	5.86	14.67	159.55	36.45
	VPP-4	6.86	0.00	57.20	20.43
	VPP-10	8.38	0.00	51.63	19.63
	Linux	7.56	54.60	12835.1	588.1

many concurrent state entries. NAT was implemented using *nftables* in Linux, and the *nat_plugin* in VPP.

A one-way traffic pattern is used to isolate the performance impact of NAT translations without interference from return traffic, as response packets would be subject to the same translation and state tracking processes, merely in the opposite direction.

2.3.3.1 1 Gbps Test Results

As the results in Table 2.11 show, the introduction of NAT had a slight impact on all VPP configurations compared to plain one-way forwarding at 1 Gbit/s. All VPP configurations also exhibited a consistent packet loss of approximately 1.5%. Since this behavior is present across all test cases, it suggests that VPP may drop the first few packets of each flow until a NAT session table entry is established. The stateless UDP traffic generated by TRex provides very limited context for session tracking, which likely contributes to this issue. In real-world network traffic – typically bidirectional and stateful – this behavior would likely not occur or would be mitigated by application-layer retries. Interestingly, Linux delivered all packets with low latency except in the 64-byte frame test. The slightly better results of Linux with NAT compared to plain forwarding may be explained by the presence of precomputed conntrack sessions, which can accelerate routing decisions in the kernel.

Table 2.11 Results of NAT 1 Gbit/s tests

	Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
64B	VPP-1	5.67	1.56	19.10	11.65
	VPP-4	6.37	1.56	35.75	16.35
	VPP-10	7.98	1.56	31.60	15.20
	Linux	6.61	4.43	303.25	182.85
512B	VPP-1	5.76	1.53	12.25	12.25
	VPP-4	6.36	1.53	26.60	19.30
	VPP-10	7.90	1.53	25.60	18.25
	Linux	6.21	0.00	18.50	10.25
889B	VPP-1	5.74	1.50	10.30	7.45
	VPP-4	6.26	1.50	23.40	18.40
	VPP-10	7.81	1.51	23.80	20.10
	Linux	6.13	0.00	12.30	11.30
1280B	VPP-1	5.73	1.48	7.70	7.30
	VPP-4	6.35	1.48	18.45	15.45
	VPP-10	7.75	1.48	19.10	16.45
	Linux	6.07	0.00	11.40	1.30
1518B	VPP-1	5.68	1.47	7.30	5.55
	VPP-4	6.35	1.47	17.55	16.50
	VPP-10	7.86	1.47	17.5	15.85
	Linux	6.05	0.00	10.55	6.60

The energy efficiency graph shown in Fig. 2.10 demonstrates results similar to those observed in plain one-way 1 Gbit/s forwarding.

2.3.3.2 10 Gbps Test Results

In this high-traffic scenario, the results presented in Table 2.12 show that the increased packet rate had a noticeable impact on the Linux stack – particularly in tests with smaller frame sizes – resulting in significantly worse performance compared to plain 10 Gbit/s one-way forwarding. This highlights the considerable computational overhead introduced by NAT when combined with packet forwarding, especially at high packet-per-second rates. VPP also exhibited some performance degradation in the 64-byte frame test compared to forwarding-only test. However, in all other cases, VPP maintained performance close to the plain forwarding baseline, with only minor differences. These results clearly demonstrate VPP's superior packet processing efficiency over the Linux network stack under heavy NAT workloads.

The energy efficiency graph in Fig. 2.11 shows results that closely mirror those observed in the plain one-way 10 Gbps forwarding test. The only notable

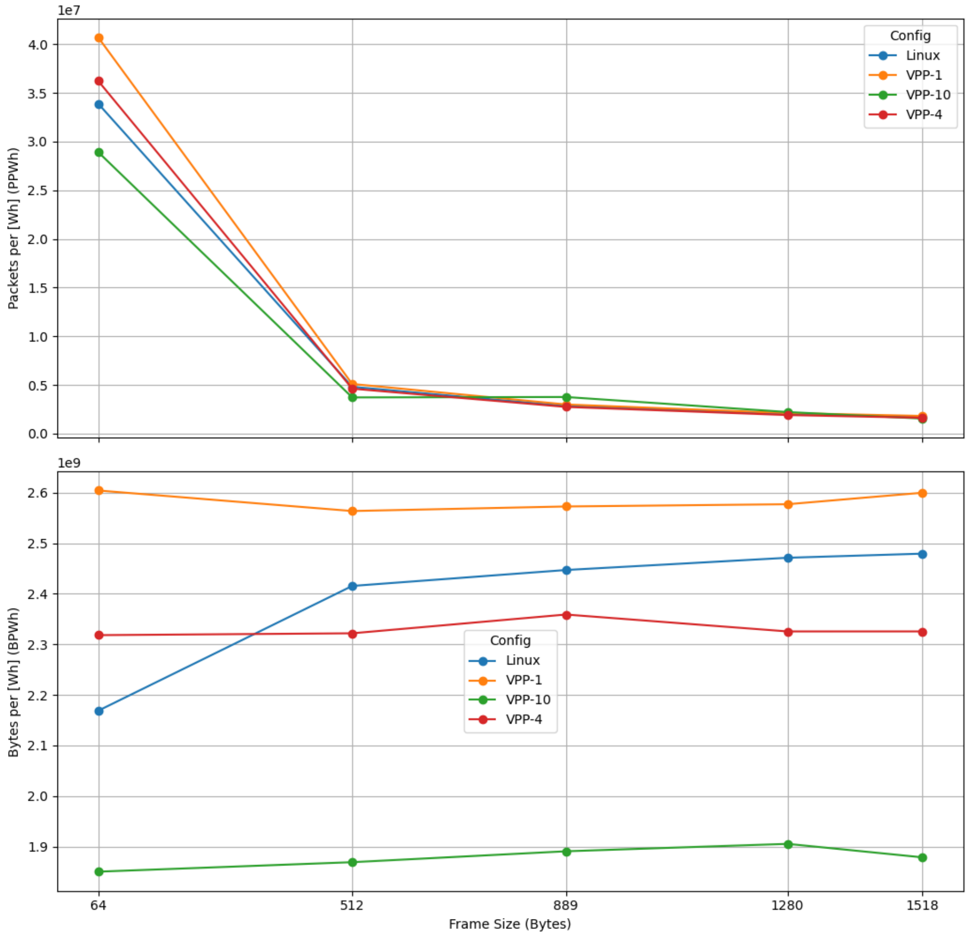

Figure 2.10 Energy efficiency per delivered data in NAT 1 Gbit/s

Table 2.12 Results of NAT 10 Gbit/s tests

	Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
64B	VPP-1	5.75	75.12	259.15	11.95
	VPP-4	6.17	51.98	735.00	55.20
	VPP-10	7.95	5.86	676.15	76.15
	Linux	7.25	84.03	10314.40	1683.40
512B	VPP-1	5.56	1.56	31.15	16.90
	VPP-4	6.20	1.56	40.10	17.45
	VPP-10	7.76	1.56	32.40	17.65
	Linux	6.87	9.02	255.10	177.45
889B	VPP-1	5.57	1.56	29.80	18.55
	VPP-4	6.28	1.56	34.45	21.60
	VPP-10	7.78	1.56	29.45	17.00
	Linux	6.63	0.00	166.20	129.60
1280B	VPP-1	5.66	1.55	26.65	18.10
	VPP-4	6.26	1.55	31.50	19.50
	VPP-10	7.79	1.56	28.55	17.20
	Linux	6.49	0.00	101.20	93.40
1518B	VPP-1	5.64	1.55	26.10	18.20
	VPP-4	6.21	1.55	30.15	19.35
	VPP-10	7.88	1.55	26.55	18.60
	Linux	6.45	0.00	67.25	61.60

deviation is the reduced performance of the VPP-4 configuration in the 64-byte frame scenario.

2.3.3.3 25 Gbps Test Results

As shown in Table 2.13, even the VPP-10 configuration dropped more than half of all packets in the 64-byte frame scenario. All VPP configurations experienced a noticeable spike in packet loss rates, particularly in the 64-byte frame test. The results in this test are more comparable to the 40Gbit/s basic forwarding scenario than to the 25Gbit/s one. In the remaining frame size tests, all VPP latency metrics approximately doubled compared to basic 25 Gbit/s forwarding. The Linux configuration, once again, struggled with high latency and severe packet loss. Overall, these findings further demonstrate VPP's capability to handle high-throughput traffic – although NAT processing introduces a significant computational burden.

The energy efficiency graph in Fig. 2.12 shows results broadly similar to those observed in the plain one-way 25,Gbps forwarding scenario. However, the Linux configuration performed slightly worse in terms of BPWh and PPWh, falling behind the VPP-10 configuration in the 889-byte frame scenario.

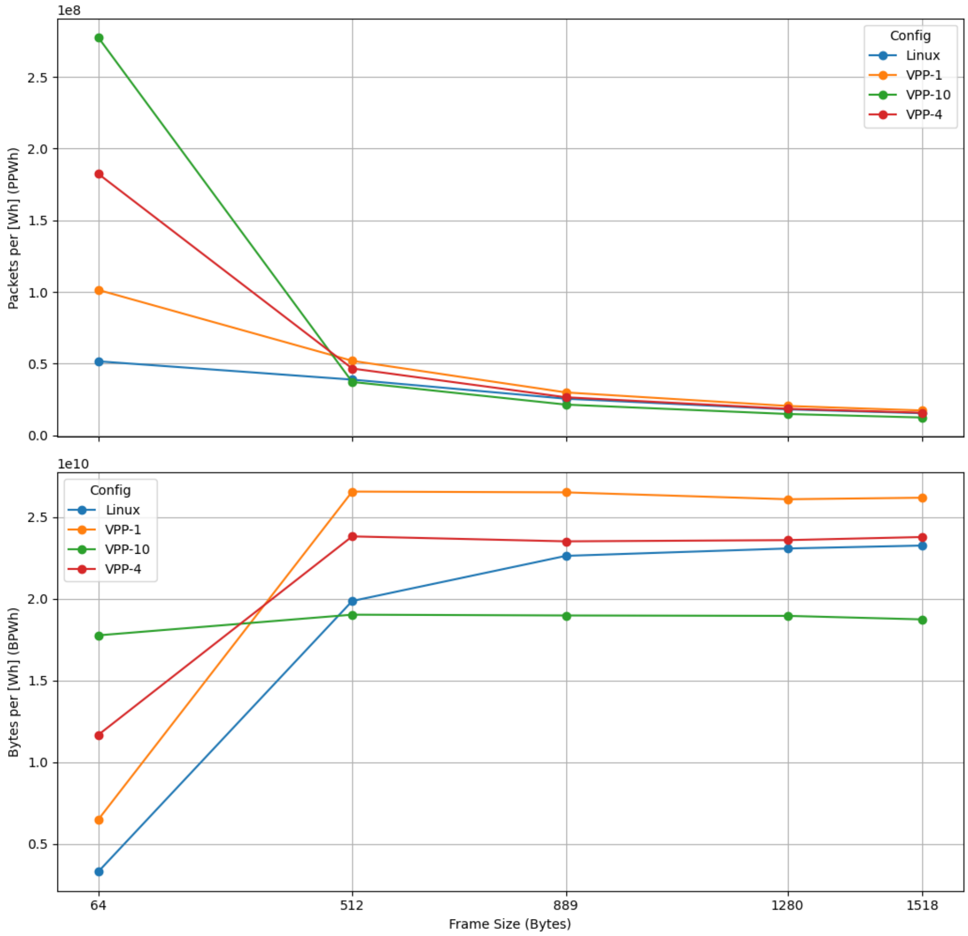

Figure 2.11 Energy efficiency per delivered data in NAT 10 Gbit/s

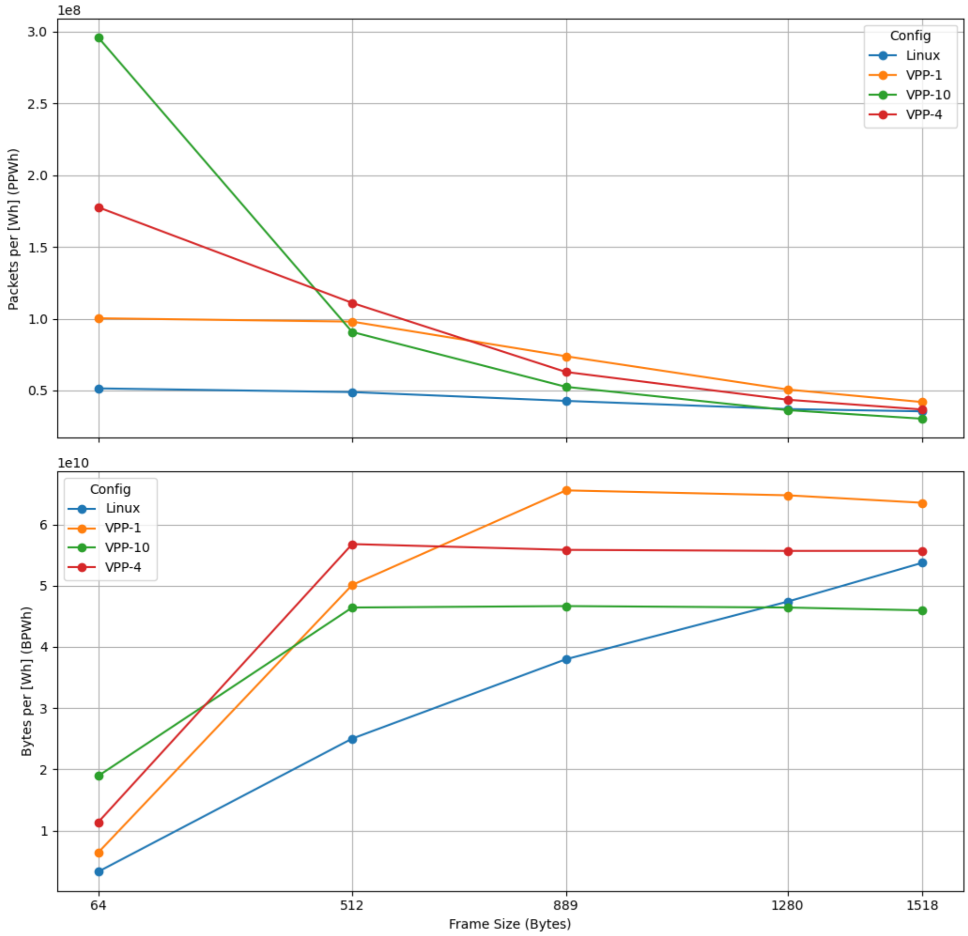

Figure 2.12 Energy efficiency per delivered data in NAT 25 Gbit/s

Table 2.13 Results of NAT 25 Gbit/s tests

	Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
64B	VPP-1	5.73	90.19	459.65	17.80
	VPP-4	6.39	80.63	836.1	58.95
	VPP-10	7.95	59.86	851.35	57.60
	Linux	7.26	93.62	8398.85	1636.1
512B	VPP-1	5.70	23.82	256.05	23.55
	VPP-4	6.50	1.56	71.55	23.05
	VPP-10	7.95	1.56	40.05	15.85
	Linux	7.30	51.35	4566.95	408.6
889B	VPP-1	5.63	1.57	44.10	30.15
	VPP-4	6.61	1.56	47.45	21.40
	VPP-10	7.91	1.56	35.70	21.30
	Linux	7.02	28.91	6797.05	403.45
1280B	VPP-1	5.70	1.56	35.10	15.95
	VPP-4	6.63	1.56	43.05	21.60
	VPP-10	7.95	1.56	36.70	18.20
	Linux	6.92	12.54	262.25	204.50
1518B	VPP-1	5.81	1.56	31.20	14.40
	VPP-4	6.63	1.56	40.65	22.50
	VPP-10	8.03	1.56	34.40	18.45
	Linux	6.82	2.25	282.15	175.4

2.3.3.4 40 Gbps Test Results

The results of the most extreme NAT test presented in Table 2.14 continue the trend observed in the previous test. In scenarios where VPP handled almost the full processing of traffic, the average latency in some cases nearly doubled compared to pure forwarding in the 40 Gbit/s test.

As shown in the energy efficiency graph in Fig. 2.13, Linux was the least energy-efficient option, performing worse than any of the VPP configurations. Even VPP-1 showed slightly lower efficiency compared to the other VPP setups, and in the 889-byte frame scenario, its performance dropped below that of VPP-4.

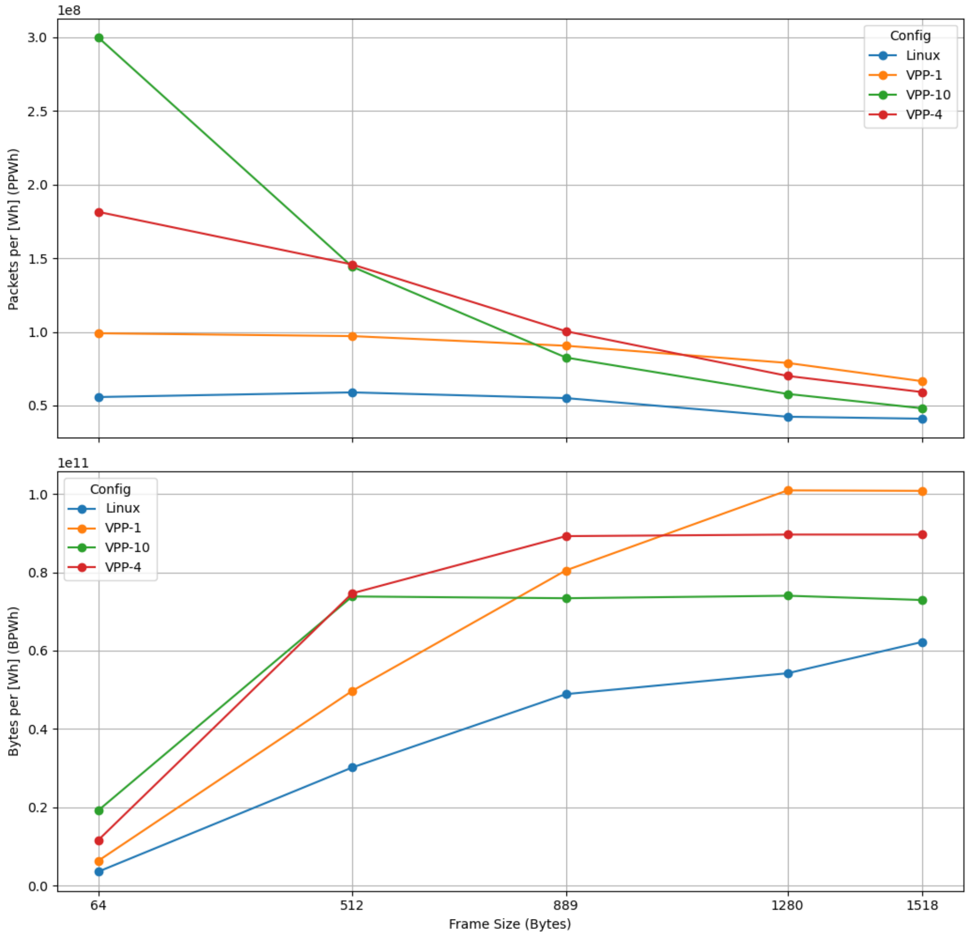

Figure 2.13 Energy efficiency per delivered data in NAT 40 Gbit/s

Table 2.14 Results of NAT 40 Gbit/s tests

	Config	Energy [Wh]	Pkt Loss [%]	Avg Lat [μs]	Jitter [μs]
64B	VPP-1	5.78	93.89	447.95	16.60
	VPP-4	6.41	87.59	830.80	44.80
	VPP-10	7.98	74.47	860.45	55.85
	Linux	7.21	95.71	3507.75	456.25
512B	VPP-1	5.79	52.02	264.75	18.45
	VPP-4	6.61	17.80	793.65	64.10
	VPP-10	8.00	1.56	49.75	17.15
	Linux	7.32	63.20	3017.60	287.65
889B	VPP-1	5.82	21.93	324.95	33.50
	VPP-4	6.62	1.56	65.25	22.60
	VPP-10	8.05	1.56	44.10	17.90
	Linux	7.43	39.44	2838.95	127.05
1280B	VPP-1	5.85	1.61	71.20	24.95
	VPP-4	6.59	1.56	51.60	21.25
	VPP-10	7.98	1.56	41.60	19.95
	Linux	7.15	35.39	5840.10	313.60
1518B	VPP-1	5.86	1.57	48.65	26.90
	VPP-4	6.59	1.56	47.80	20.60
	VPP-10	8.10	1.56	39.00	19.45
	Linux	7.09	26.47	7408.65	350.9

2.4 Key Observations from Results

The evaluation focused on throughput, latency, packet loss, and energy efficiency under different traffic scenarios and system configurations.

As demonstrated by the overall results, the Linux networking stack generally performs better under lighter loads. When the packet rate is sufficiently low, the interrupt-based approach proves to be more efficient. Moreover, under low-traffic conditions, VPP tends to exhibit higher latency, especially when multiple receive queues are configured. This may be attributed to VPP's limited ability to fully leverage the benefits of vectorized packet processing at low volumes. When a graph node is triggered – a relatively costly operation – incoming packets may have to wait for the current batch to complete. Enabling Receive Side Scaling in these scenarios can further exacerbate the issue, as limited traffic is distributed across multiple queues and workers, leading to unnecessary overhead. These observations proved that VPP is not optimized for handling sparse or low-throughput traffic patterns.

On the other hand, when handling higher traffic volumes, VPP processes more fully populated vectors and can take full advantage of vectorization optimizations, which helps amortize the per-packet processing cost. As demonstrated, under heavy load, VPP is able to maintain reasonable latency, while the Linux stack becomes flooded with system calls, and its per-packet processing model reveals clear performance limitations. This is further supported by the observed packet loss in VPP, particularly under heavy NAT workloads with small frame sizes. It was also observed that VPP's parallelism become truly apparent only under high-throughput conditions.

A serious drawback of VPP is that it consumes nearly the same amount of energy even when there is no traffic to process. While this is less noticeable under typical load conditions, it becomes highly inefficient during periods of minimal or no traffic – such as early morning hours in enterprise or residential networks – where VPP continues polling the NIC despite having nothing to handle. VPP does not support on-the-fly hardware configuration changes and typically requires a full restart; a simple reload is not sufficient. This further worsens its suitability for dynamic or energy-sensitive environments, where the ability to adjust to changing load conditions – such as reduced traffic during off-peak hours – is essential. In contrast, the Linux networking stack can accommodate such changes at runtime, allowing CPUs to be enabled, disabled, or reassigned without interrupting network services.

It is important to note that these measurements were conducted using synthetic traffic in a controlled environment, designed to evaluate how VPP and the Linux networking stack behave under specific conditions. Real-world traffic patterns, including bursts and mixed flow types, may result in different performance characteristics.

Chapter 3

Conclusion and Future Work

The objective of this thesis was to analyze and compare the performance, latency, and energy efficiency of different VPP configurations with a traditional Linux-based router. The work aimed to explore the theoretical advantages of VPP, design and implement a custom measurement infrastructure, define relevant traffic scenarios, and determine the conditions under which VPP outperforms the Linux network stack.

The theoretical part of the thesis provided a detailed explanation of VPP's architecture and operating principles. It also included a comparative analysis of packet processing between VPP and the traditional Linux network stack, highlighting the key differences in design, performance potential, and resource utilization.

As part of the practical work, a custom measurement infrastructure was designed and implemented to enable reliable and repeatable testing. Using this setup, a total of 4800 measurements, reflecting realistic usage conditions on standard COTS hardware, were conducted across 60 different test scenarios using four configurations – three VPP variants and one Linux-based reference system. This resulted in 240 distinct test cases, providing a robust dataset for evaluating throughput, latency, and energy efficiency under a variety of traffic conditions.

The results show that VPP consistently outperformed the Linux-based reference system in terms of throughput and latency – with the exception of low-traffic forwarding scenarios involving larger frame sizes, where the Linux stack achieved slightly better results. However, these differences were relatively minor. The Linux-based reference system was more energy-efficient than all VPP configurations only under light traffic conditions – and with the exception of VPP with a single worker thread. VPP, however, demonstrated more consistent performance in terms of throughput and latency across all traffic conditions. Given VPP's generally higher performance ceiling, it is reasonable to expect it would handle sudden traffic spikes more effectively and with greater consistency than the Linux-based solution. The results also clearly indicate that VPP would be more resilient under DoS or DDoS-style attack conditions.

Several directions for future work remain open. For small-to-mid-sized ISPs, it would be useful to compare VPP not only with the Linux network stack, but also with commercial or legacy routers commonly used in practice. It would also be valuable to evaluate the impact of advanced features such as Quality of Service (QoS), firewalling, or traffic shaping. Finally, comparing VPP with a custom-tuned Linux kernel optimized for networking could provide deeper insight into the limits of traditional software-based routing.

Appendix A
TRex Measurement Profile

This appendix contains the profile used by the TRex traffic generator during all performance tests presented in this thesis. The profile defines two streams. The first, called *heavy*, represents the main traffic stream with varying frame sizes. Its source IP address is incremented to generate distinct packets, simulating a more realistic traffic pattern. The second stream, named *latency*, is used for latency measurements. It uses 64-byte frames and static IP addresses, and its transmission rate is set to 5000 packets per second. In bidirectional scenarios, a mirrored version of this profile was deployed on the second interface to generate traffic in the opposite direction, using a distinct `pg_id` to enable independent latency measurement.

```
from trex_stl_lib.api import *

class STLS1(object):
    def __init__(self):
        self.fsize = [FRAME_SIZE] # Frame size for the main
            traffic stream (e.g., 64, 512... bytes)
        self.latency_fsize = 64 # Frame size for latency
            stream
        self.cache_size = [CACHE_SIZE] # Max number of
            unique source IPs set in STLVmFlowVar
            (precomputed if set); can be 'None'

    def create_heavy_stream(self):
        size = self.fsize - 4 # FCS
        base_pkt = Ether()/IP(src='16.0.0.1',
            dst='48.0.0.1')/UDP(dport=12, sport=1025)
        pad = max(0, size - len(base_pkt)) * 'x'

        vm = STLScVmRaw([
             STLVmFlowVar("ip_src", min_value="10.0.0.1",
                 max_value="10.255.255.255", size=4, step=1,
                 op="inc"),
```

```
            STLVmWrFlowVar(fv_name="ip_src", pkt_offset="IP.src"),
            STLVmFixIpv4(offset="IP") # Fix IPv4 header checksum
            STLVmFixChecksumHw(l3_offset="IP", l4_offset="UDP", l4_type=17) # Fix UDP checksum
        ], cache_size=self.cache_size)

        pkt = STLPktBuilder(pkt=base_pkt/pad, vm=vm)

        return STLStream(packet=pkt, mode=STLTXCont())

    def create_latency_stream(self):
        size = self.latency_fsize - 4 # FCS
        pkt = Ether()/IP(src="16.0.0.1", dst="48.0.0.1")/UDP(dport=1234, sport=1234)
        pad = max(0, size - len(pkt)) * 'x'
        pkt = STLPktBuilder(pkt=pkt/pad)

        return STLStream(
            packet=pkt,
            mode=STLTXCont(pps=5000), # Packet rate for latency stream
            flow_stats=STLFlowLatencyStats(pg_id=7) # Stream ID for latency statistics
        )

    def get_streams(self):
        return [self.create_heavy_stream(), self.create_latency_stream()]

def register():
    return STLS1()
```

Code listing A.1 TRex Measurement Profile Example

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?* [online]. 2025. [visited on 2025-04-07]. Available from: `https://wiki.fd.io/view/VPP/What_is_VPP%3F`.
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* [online]. 2021. [visited on 2025-04-10]. Available from: `https://fd.io/docs/vpp/v2101/whatisvpp/extensible`.

7. FD.IO PROJECT. *Multi-threading in VPP* [online]. 2025. [visited on 2025-05-03]. Available from: https://s3-docs.fd.io/vpp/25.06/developer/corearchitecture/multi_thread.html.
8. DPDK PROJECT. *About DPDK* [online]. 2025. [visited on 2025-04-13]. Available from: https://www.dpdk.org/about/.
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* [online]. 2024. [visited on 2025-04-20]. Available from: https://core.dpdk.org/supported/nics/.
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* [online]. 2023. [visited on 2025-04-19]. Available from: https://docs.kernel.org/networking/scaling.html.
14. INTEL CORPORATION. *Core Utilization in DPDK Applications* [online]. 2024. [visited on 2025-04-19]. Available from: https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookbook/2025-0/core-utilization-in-dpdk-apps.html.
15. INTEL CORPORATION. *PCIe Traffic in DPDK Applications* [online]. 2024. [visited on 2025-04-19]. Available from: https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookbook/2025-0/pcie-traffic-in-dpdk-apps.html.
16. FD.IO PROJECT. *VPP Software Architecture* [online]. 2025. [visited on 2025-04-21]. Available from: https://s3-docs.fd.io/vpp/25.06/developer/corearchitecture/softwarearchitecture.html.
17. FD.IO PROJECT. *VPP Infrastructure Layer* [online]. 2025. [visited on 2025-04-21]. Available from: https://s3-docs.fd.io/vpp/25.06/developer/corearchitecture/infrastructure.html.
18. FD.IO PROJECT. *VNET - VPP Network Stack* [online]. 2025. [visited on 2025-04-21]. Available from: https://s3-docs.fd.io/vpp/25.06/developer/corearchitecture/vnet.html.

19. FD.IO PROJECT. *Feature Arcs* [online]. 2025. [visited on 2025-04-21]. Available from: `https://s3-docs.fd.io/vpp/25.06/developer/corearchitecture/featurearcs.html`.
20. *VLIB Architecture — VPP Developer Documentation (Release 25.06)* [online]. FD.io Project, 2025. [visited on 2025-04-21]. Available from: `https://s3-docs.fd.io/vpp/25.06/developer/corearchitecture/vlib.html`.
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* [online]. 2025. [visited on 2025-05-03]. Available from: `https://s3-docs.fd.io/vpp/25.06/aboutvpp/supported.html`.
24. FD.IO PROJECT. *FD.io release package repository on Packagecloud.io* [online]. 2025. [visited on 2025-05-03]. Available from: `https://packagecloud.io/fdio`.
25. FD.IO PROJECT. *Building VPP* [online]. 2025. [visited on 2025-05-03]. Available from: `https://s3-docs.fd.io/vpp/25.06/developer/build-run-debug/building.html`.
26. FD.IO PROJECT. *Running VPP* [online]. 2025. [visited on 2025-05-03]. Available from: `https://s3-docs.fd.io/vpp/25.06/gettingstarted/running/index.html`.
27. DPDK PROJECT. *Linux Drivers* [online]. 2025. [visited on 2025-05-03]. Available from: `https://doc.dpdk.org/guides/linux_gsg/linux_drivers.html`.
28. FD.IO PROJECT. *Getting Started with the Configuration* [online]. 2025. [visited on 2025-05-03]. Available from: `https://s3-docs.fd.io/vpp/25.06/configuration/config_getting_started.html`.
29. FD.IO PROJECT. *Configuration Reference* [online]. 2025. [visited on 2025-05-03]. Available from: `https://s3-docs.fd.io/vpp/25.06/configuration/reference.html`.
30. FD.IO PROJECT. *VPP Feature List – FD.io Documentation (Release 25.02)* [online]. 2025. [visited on 2025-04-15]. Available from: `https://s3-docs.fd.io/vpp/25.02/aboutvpp/featurelist.html`.
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* [online]. 2025. [visited on 2025-04-16]. Available from: `https://iperf.fr/`.

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* [online]. 2025. [visited on 2025-04-16]. Available from: `https://trex-tgn.cisco.com/`.

37. PROJECT, DPDK. *Pktgen-DPDK* [online]. 2025. [visited on 2025-04-16]. Available from: `https://pktgen-dpdk.readthedocs.io/`.

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

39. 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`.

40. IEEE Standard for Ethernet. *IEEE Std 802.3-2022 (Revision of IEEE Std 802.3-2018)*. 2022, pp. 1–7025. Available from DOI: `10.1109/IEEESTD.2022.9844436`.

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

Contents of the attachment

```
/
├── readme.txt ........................ brief description of the attachment
├── src
│   ├── measurements ...................... collected measurement outputs
│   └── thesis .......................... source code of the thesis in LaTeX
└── thesis.pdf ............................... thesis text in PDF format
```