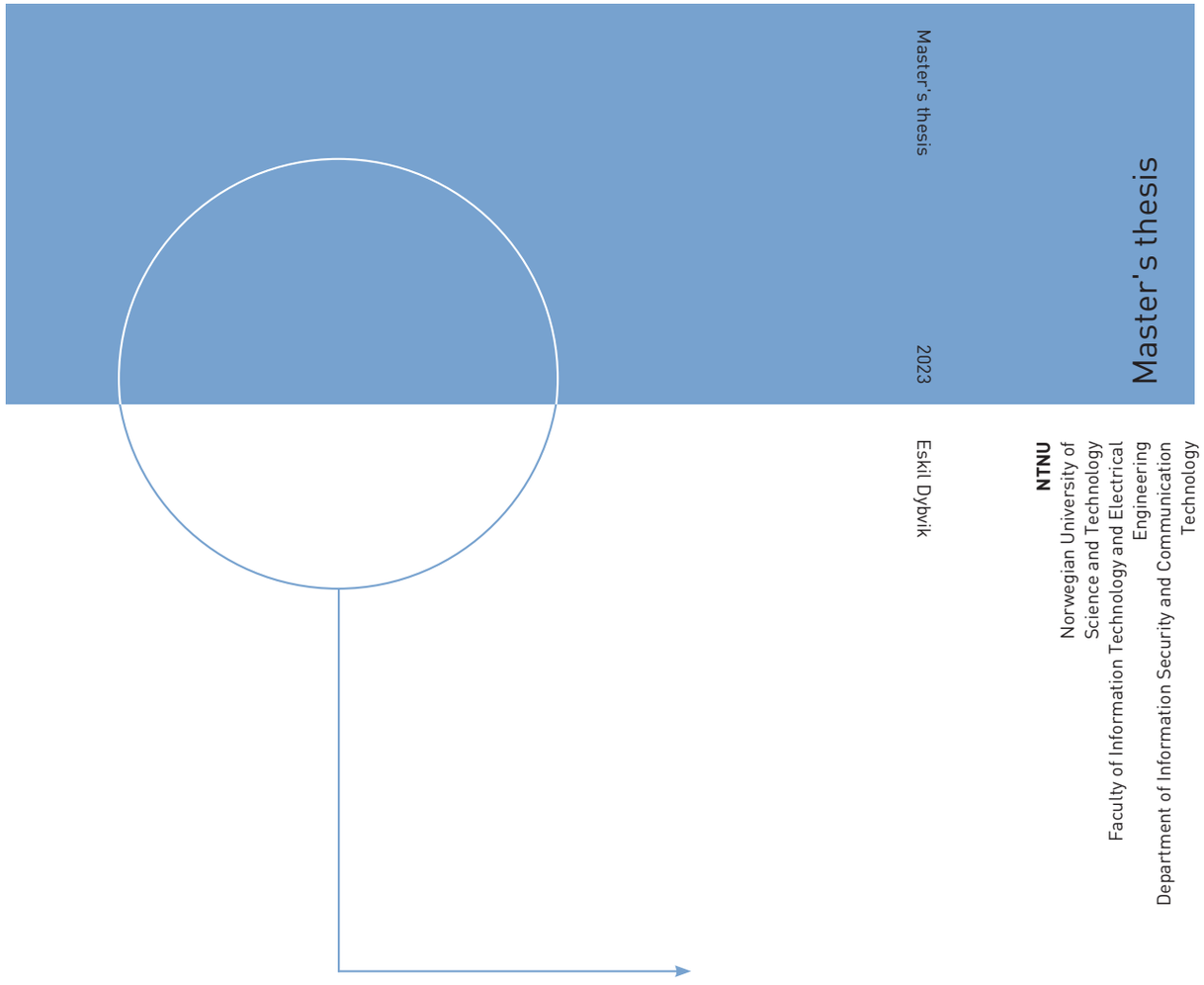

Master's thesis

2023

Eskil Dybvik

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Information Security and Communication Technology

Eskil Dybvik

Open source benchmarking of the Linux kernel network stack for real-time applications

June 2023

NTNU
Norwegian University of Science and Technology

NTNU
Norwegian University of Science and Technology

NTNU
Norwegian University of
Science and Technology

Open source benchmarking of the Linux kernel network stack for real-time applications

Eskil Dybvik

MTKOM
Submission date: June 2023
Supervisor: David Palma

Norwegian University of Science and Technology
Department of Information Security and Communication
Technology

Title: Open source benchmarking of the Linux kernel network stack for real-time applications

Student: Dybvik, Eskil

Problem description:

The world around us is getting more and more interconnected, and many systems are evolving into "smart" systems. Examples of this development, are the emergence of smart power grids and remote surgery. These systems rely on fast and stable connection between devices in order to quickly react to events. Many systems rely on the TCP/IP stack for this communication. However, TCP/IP was designed as a "best-effort" network, without offering any guarantees for packet delivery time or stability.

This thesis aims to develop an open source benchmarking testbed to systematically verify if a device is capable of handling latency- and jitter-sensitive real-time applications. Unlike other solutions, this testbed will not just inspect averages and aggregated values, but rather look at patterns in the individual traffic flows, to determine potential problems for a time-sensitive application. The testbed will use readily available off-the-shelf hardware, and open source software to allow anyone to reconstruct the testbed to benchmark and validate their own devices.

After developing the testbed, the thesis aims to use this to benchmark and investigate the performance of the network stack of the Linux kernel. Linux is a widely used operating system, with a high degree of flexibility and configurability. The thesis will systematically try out various configurations parameters of the kernel, and determine their importance for real-time applications.

Approved on: 2023-03-22

Supervisor: Palma, David, NTNU

Abstract

In the context of time-sensitive networking, such as smart power grids and remote controlled machinery, it is important to minimize the sources of latency and jitter in the network. In this thesis, we investigate how changing various parameters of the Linux network stack may affect the performance of a network device. Traditionally, related work focus on aggregated values such as average latency and jitter, but we argue that it is important to also look at the extreme outliers in the data.

To perform tests systematically, we have developed a benchmarking testbed solution for testing a network device's suitability for real-time applications. The testbed uses a modified version of Cisco TRex to generate traffic, and a custom data processing tool to analyze the results. We define a metric called "anomalies", defined as *n consecutive packets with a latency higher than t*. The testbed is designed to be easily reproducible, and to be able to run on a wide range of hardware, making it useful for potential future research.

Using the testbed, we have performed a series of experiments to investigate how changing various parameters of the Linux network stack affects the performance of a network device. We tested different queue sizes, different traffic rates, different system loads, tested with the `PREEMPT_RT` kernel, and with applying a recent patch that changes the way the Linux kernel handles network interrupts. We found that there are combinations of parameters that perform much better than others, but that there is no single configuration that fits all use cases.

Sammendrag

Når vi snakker om tidskritiske nettverksenheter, som for eksempel i smarte strømnett eller fjernstyrte kjøretøy, er det viktig å minimere kilder til forsinkelse og variasjoner i denne. I denne oppgaven har vi sett på hvordan nettverksstakken til Linux blir påvirket når man endrer på diverse parametre og verdier, og hvordan dette påvirker ytelsen til nettverksenheter. Tidligere forskning baserer seg hovedsakelig på sammenlagte verdier, som for eksempel gjennomsnittlig forsinkelse og varians, mens vi i denne oppgaven heller fokuserer på de sjeldnere ekstremverdiene.

For å gjøre dette på en systematisk måte, har vi utviklet en testløsning for nettverksenheter, med fokus på tidssensitive applikasjoner. Testløsningen baserer seg på å bruke en modifisert versjon av Cisco TRex til å generere nettverkstrafikk, og et selvlaget program for å tolke dataen som blir generert. Vi definerte et mål vi kaller «anomalies», som vi har definert som *n påfølgende pakker med en forsinkelse som er høyere enn t*. Testoppsettet er laget for å være gjenbrukbart og støtte et bredt utvalg av forskjellig maskinvare, og vil derfor kunne brukes i fremtidig forskning.

Ved å bruke testløsningen vår, gjorde vi en rekke tester på Linux-kjernens nettverksstakk. Vi eksperimenterte med å variere størrelsen på køene til nettverkskortet, forskjellige mengder nettverkstrafikk, testet med `PREEMPT_RT`-kjernen, og ved å teste ut en relativt ny patch til kjernen som lar brukeren endre hvordan kjernen håndterer interrupts fra nettverkskortet. Gjennom testingen fant vi ut at det er store variasjoner mellom de ulike kombinasjonene av parametre, som viser at det kan være mye å hente på å optimalisere kjernen for spesifikke applikasjoner. Det var ingen parametre som viste seg å universelt forbedre ytelsen, så det vil være nødvendige å gjøre tester for hver enkelt applikasjon.

Preface

This report is the result of my master's thesis in communication technology and digital security at the Norwegian University of Science and Technology (NTNU). It was conducted in the spring of 2023. All the source code for tools developed as part of the thesis is available at my GitHub account, at https://github.com/KHTangent/thesis-subprojects.

This thesis would not have been possible without the help and support from my supervisor David Palma, who provided a lot of knowledge and guidance on formalizing the project. I would like to thank him for his help, and for always being available for questions. I would also like to thank Hanoch Haim at Cisco for developing and supporting the TRex traffic generator, which was used a lot during the thesis. I would like to thank the Department of Information Security and Communication Technology at NTNU for providing great hardware resources for the project. In addition, I would like to thank Zolve AS, for hiring me and give me a break from the thesis work when needed, and to Karin at Zolve, for making delicious lunch every day. Lastly, a big thanks to my family and friends for supporting me throughout the project.

Writing a master's thesis has been a challenging, but also rewarding experience. I have learned a lot more about network performance, and how to do systematic testing of network devices. I have also learned a lot about how to write a scientific report, and how to structure a large project. In the end, I hope that the work I have done can be useful for future researchers, and that it can be used as a starting point for further research.

Contents

List of Figures

List of Tables

List of Acronyms

CLI Command Line Interface.

COTS Commercial off-the-shelf.

CPU Central Processing Unit.

DMA Direct Memory Access.

DPDK Data Plane Development Kit.

DUT Device Under Test.

NIC Network Interface Card.

PCAP Packet Capture.

PGEN Packet Generator.

RFC Request For Comments.

SSD Solid State Drive.

TCP Transmission Control Protocol.

Chapter 1
Introduction

Major parts of this introduction are based on the preliminary work done in the pre-project for the thesis, which was conducted during the fall of 2022[12].

1.1 Motivation

The world is getting more interconnected, and more devices than ever are being connected to the internet. From smartphones, to children's toys, appliances, and critical infrastructure, the amount of devices with some form of internet connection is increasing rapidly. Many of these devices run a variant of the Linux operating system[19].

There are many reasons for choosing Linux for an internet-connected device. It is widely used, so many resources are available. It is free and open source, and can be tailored for almost any kind of workload. Configuration parameters can be applied both at runtime, and while configuring the source code before compiling. This gives system administrators and device manufacturers many ways to make sure their installations work in an optimal way. However, this flexibility has its downsides. How can a user know which options matter, and what to set them to? Another aspect that might affect how well a system works, is the Linux kernel itself. Linux is a large piece of software, with many interconnected parts. This gives plenty of room for errors and poorly optimized code. Are there any parts of the kernel that would benefit more than others from some optimization?

An example of optimizations working well, can be seen in the FreeBSD Journal[34]. FreeBSD is, like Linux, a Unix-like operating system. The article looked into the performance of the software network bridge in the FreeBSD kernel, and discovered that the bridge was not as fast as expected. By using various tracing tools, the author found that most of the Central Processing Unit (CPU) time during bridging was spent on waiting for software locks. By rewriting parts of the code to use more modern semaphore implementations instead of simple locks, the performance of the

network bridge was improved by about 500%. Could anything similar exist in the Linux kernel?

1.2 Methodology

The thesis has two distinct main parts: "The testbed part" and "The evaluation part". The testbed part will consist of developing and documenting a proper testbed setup for real-time network applications, while the evaluation part will use the created testbed to evaluate the importance of various configuration parameters in the Linux kernel.

1.2.1 Design science methodology

The first part of the thesis, is a typical design science problem. A design science problem differs from the more widely known natural science problem by having a goal of creating a solution to a problem, instead of explaining a phenomenon[25]. One approach that can be used to create a solution to a design science problem, is to use the design science methodology. The main part of design science methodology, is the design cycle. For a thesis like ours, the cycle will have three steps, which are repeated until a solution has been created.

Step 1: Problem Investigation In this stage, the various requirements and stakeholders involved in the problem are identified. This includes finding out why the problem exists, and what would be needed to solve it.

Step 2: Treatment Design In design science, "treatment" refers to a solution to a problem. The second stage of the design cycle, is about developing a potential solution to a problem, based on the requirements specified in the first step.

Step 3: Treatment Validation Once a potential treatment has been developed, it is time to test and verify it. If the treatment is found to solve the problem, the design cycle is complete. Otherwise, the cycle is restarted from step 1, but with new knowledge obtained from the previous iteration.

1.2.2 Part 1: The testbed part

In the first part of the thesis, we will continue on the background study from the pre-project[12], in order to get an understanding of the requirements for a testbed. This corresponds to the first step of the design cycle described in the previous subsection.

After landing on some requirements for the testbed, we can begin the design process. This will include both software setup, hardware setup, and perhaps most importantly documentation. The documentation is one of the main artifacts from the thesis, as this will allow anyone to recreate the testbed to perform their own experiments. We will write a draft for the documentation while developing the testbed, to make sure all details are included, before improving it when the design is complete.

While working on the testbed, we will conduct small intermediate tests to get an understanding of how well the setup works. The results obtained at this stage will only be used for validation of the testbed setup, and will not be used for the evaluation part of the thesis. These intermediate tests will include as few variables as possible, for example by running the tests in loopback mode (explained in subsection 3.3.1).

Completing this part of the thesis is vital for starting the second part, the evaluation part. This can be seen as a treatment evaluation of the developed testbed.

1.2.3 Part 2: The evaluation part

In the evaluation part, we will repeatedly use the testbed from part 1 of the thesis. The testbed will be used to try out the effects of applying various configuration parameters to the Linux kernel. To get a sense of stability in the results, we will perform each test several times, and with many different permutations of configuration parameters. For efficiency, we will use a batching script for this, so minimal user interaction will be required.

Before starting the tests, we will create a plan, with a list of what to test, how to test it, and expected results. After we have obtained results, we will analyze them, and make attempts at drawing conclusions from the obtained results.

1.3 Research questions

While doing the pre-project for this thesis[12], some research questions were defined. These will be carried over to the thesis, so the questions and justifications from the pre-project are included below.

RQ1: How can we create an open-source benchmarking testbed for network performance assessments?

RQ2: What are the performance bottlenecks in the Linux network stack, in the context of real-time applications running on a desktop server?

RQ3: What is the optimal configuration of a Linux server used for real-time applications?

RQ1 is the main research question for the testbed part of the thesis. If we can get the testbed to work as intended, we will have an answer to this question. In addition, the testbed will help us answer the other two research questions.

RQ2 and **RQ3** are the main research questions for the evaluation part of the thesis. These questions will be answered by using the testbed to perform several tests with different configurations. The results from these tests will be analyzed, and used to draw conclusions about the performance of the Linux network stack, and what configuration parameters are most important for real-time applications. The questions are linked together, in that we expect the answer to **RQ2** to provide hints on what we can conclude with in **RQ3**.

Since the network stack is a very complex piece of software, we will not be able to answer these questions in full. Instead, we will try to find some of the most important parameters, and try to find out how they affect the performance of the network stack. Our hypothesis is that since the network stack is made to be general purpose, it will not be optimized for real-time applications, making it possible to tweak it for better performance.

1.4 Project outputs

The main outputs from our thesis, will be the testbed and the documentation for it. The testbed will be described in a publicly accessible GitHub repository, with all tools and scripts needed to recreate the testbed. These outputs are produced in the first part of the thesis, and validated during the second part.

In addition to the testbed, we will also produce a report describing the results from the evaluation part of the thesis, with results and plots of results. These will be published in this document.

Chapter 2
Background

In this chapter, we will introduce some of the concepts and technologies that are relevant for the rest of the thesis. We will first provide an introduction to what network traffic is, and how it can be simulated. Then, we will take a look at the Linux kernel, before evaluating some alternatives for network traffic generation.

2.1 Network traffic

In this section, we will give an introduction to what network traffic is, how artificial traffic can be generated, and how we can leverage this to benchmark network devices.

"Network traffic" refers to data being sent over a network. This data can be anything from a simple control message to a web request or video streaming. The data is sent in packets, which are small chunks of data with headers containing information about the packet. The headers contain information such as the source and destination of the packet, and the type of data contained in the packet. This design allows for a wide variety of different types of data to be sent over the same infrastructure, which increases the scalability and durability of the network.

Different types of network traffic have different requirements for how the network should behave. Some types of traffic are very robust, and can handle both long delays, packet loss, and variations in latency. Examples of this type of traffic, are file transfers and email, where protocols such as Transmission Control Protocol (TCP) help compensate for packet loss and out-of-order packets, for example by retransmitting lost packets and performing reassembly. Other types of traffic are more sensitive, and might require strict timing guarantees from the network in order to work as intended. Examples of this type of traffic, could be the operation of remote controlled machinery, where a few out-of-order packets could be the difference between a successful operation and a catastrophic failure.

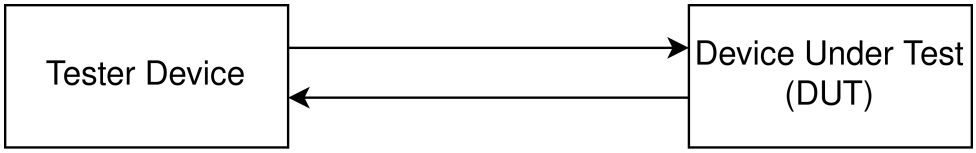

Figure 2.1: Device connections used in RFC2544

2.1.1 Benchmarking network devices

During the pre-project for this thesis, we performed a study of how network devices are benchmarked[12]. This subsection builds on the results from the study, and extends it where appropriate.

In order to measure and benchmark network performance, a good benchmarking testbed is needed. Developing a reliable and predictable testbed in a complex environment such as networking is difficult, and errors could make the results obtained inaccurate or even useless. One attempt at developing a standardized solution for performing network benchmarks, is published as RFC2544[2]. This Request For Comments (RFC) defines a series of possible tests to measure various metrics, including values to obtain and how to present the results. What most of those tests have in common, is how the network devices are being connected. They refer to the device being tested as Device Under Test (DUT). The DUT connects to one or more devices used to generate the data used for testing. An example connection with a single tester device testing a single DUT can be seen in Figure 2.1.

RFC2544 has received some criticism[22], not about the RFC itself, but on how these tests have been used. RFC2544 was intended for benchmarking network devices in isolated test environments, like in our project. However, enough users were running the tests in production environments such that an amendment to RFC2544 was published as RFC6815[3]. RFC6815 does not make any modifications to the benchmarks, but rather adds a warning to not run the tests in a production network.

2.1.2 Benchmarking software tools

For some types of benchmarks, it is enough to use simple software tools. Software tools are often portable and easy to set up compared to specialized hardware tools, in addition to usually being cheaper. However, software tools may have limited performance compared to dedicated hardware tools. We will look at a widely used tool for measuring network bandwidth, `iperf3`.

`iperf3` is a commonly used tool for measuring network performance. It is a

```
❯ iperf3 -c 192.168.1.10
Connecting to host 192.168.1.10, port 5201
[  5] local 192.168.1.2 port 57244 connected to 192.168.1.10 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec   114 MBytes   954 Mbits/sec    0    421 KBytes
[  5]   1.00-2.00   sec   114 MBytes   953 Mbits/sec    0    547 KBytes
[  5]   2.00-3.00   sec   112 MBytes   936 Mbits/sec    0    547 KBytes
[  5]   3.00-4.00   sec   112 MBytes   940 Mbits/sec    0    576 KBytes
[  5]   4.00-5.00   sec   113 MBytes   947 Mbits/sec    0    576 KBytes
[  5]   5.00-6.00   sec   112 MBytes   936 Mbits/sec    0    576 KBytes
[  5]   6.00-7.00   sec   113 MBytes   947 Mbits/sec    0    576 KBytes
[  5]   7.00-8.00   sec   112 MBytes   939 Mbits/sec    0    632 KBytes
[  5]   8.00-9.00   sec   112 MBytes   943 Mbits/sec    0    632 KBytes
[  5]   9.00-10.00  sec   112 MBytes   941 Mbits/sec    0    632 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  1.10 GBytes   944 Mbits/sec    0             sender
[  5]   0.00-10.00  sec  1.10 GBytes   941 Mbits/sec                  receiver
```

Figure 2.2: Screenshot of iperf3

command-line utility that can be used to measure the bandwidth of a network connection. It is open source, and is available for most popular operating systems[16]. iperf3 is used in several studies where network performance is measured[39, 4]. iperf3 uses a client-server architecture, where the DUT hosts a server, while a client initiates a test by connecting to the server. A screenshot of a typical iperf3 session can be seen in Figure 2.2, where the iperf3 client measures the maximum throughput of the connection to the server.

Software tools like iperf3 have several advantages. They are easy to set up and use, and provide a simple way to measure network performance. However, they are limited in the types of tests that can be performed, and in scale. Since these tools usually run on top of an operating system, they will be limited by the networking capabilities of the operating system.

2.1.3 Traffic generators

One commonly used tool while benchmarking network devices, is a traffic generator[26, 24]. A traffic generator is a device that generates, transmits, and receives artificial network traffic, which can be processed by the DUT. This allows device manufacturers and researchers to test realistic traffic scenarios within a closed and controlled environment. A typical traffic generator will have multiple network interfaces, where each one is used to either transmit or receive traffic.

A traffic generator can be used for a wide variety of tests. From simple tests like measuring the maximum throughput (similar to iperf3), to more complex tests like stress-testing firewalls and intrusion detection systems. Dedicated traffic generators are often optimized for generating larger amounts of traffic than software tools, and can be used to generate traffic at line rate.

Traditionally, commercial closed-source traffic generators have been used for benchmarking network devices[26, 24]. This approach has several disadvantages:

- Commercial traffic generators can get very expensive[31], which could hinder research activities without large budgets.
- Being closed source, it can be difficult to know exactly what is going on inside the generator. This gives less transparency into the benchmarking process, and may give uncertainties that would not be present if the inner workings of the generator could be inspected.
- As an extension of the point above, the closed approach of commercial traffic generators reduces the flexibility of the device.

Despite the disadvantages, commercial traffic generators can still be used for research projects. In 2007, Bolla et al. used a commercial traffic generator to benchmark the performance of the Linux kernel, similar to what we will do in this thesis[35]. The study used a traffic generator from Agilent Technologies (now Keysight[38]) to generate traffic at up to 1 Gbps.

Not all research projects have access to commercial traffic generators. Fortunately, in recent years, open-source traffic generator solutions have appeared. These traffic generators are mostly software-based, and use common Commercial off-the-shelf (COTS) hardware, while still achieving better performance than software-only tools like iperf3. This greatly reduces the entry barrier of using a traffic generator. Examples of open-source traffic generators, are MoonGen[14] and Cisco TRex[5]. Both of these will be described in more detail in section 2.3. But first, we will take a look at how networking works in the Linux kernel, which is required in order to understand some of the open-source traffic generators we will introduce later.

2.2 Linux networking

The Linux operating system has several interconnected parts. One major part of the kernel, is the network stack. The network stack is responsible for handling all TCP/IP related tasks, to allow applications to communicate over the internet in a portable and uniform way.

2.2.1 The Linux Network Stack

The network stack is the part of an operating system responsible for network-related tasks. On Linux, the stack includes everything between the network cards and the

application layer, including device drivers, system calls, protocol implementations, and the interfaces between them[15].

Closest to the hardware, are the device drivers for the network interface cards. Device drivers are specific for each network card, and are often written by the device manufacturers themselves. For example, the Network Interface Card (NIC) used in this thesis, the Intel X710-T2L, uses the `i40e` driver by Intel[8]. Drivers are responsible for the interaction between the kernel and the hardware. This includes initializing the hardware correctly, registering it to the kernel, reporting supported features, and handling the actual transmission and receiving of network traffic[30]. Drivers are required to implement certain functions and structures, which makes them interoperable with the rest of the kernel.

Traditionally, when a network packet arrives at the NIC, an interrupt is generated. An interrupt is a signal sent from the hardware to the CPU, which causes the CPU to stop what it is doing and handle the interrupt, before returning to what it was doing. In the past, this allowed the system to handle packets arriving at any time, but also caused a great amount of overhead when heavy traffic was present. To reduce the overhead, a new system was developed, known as the "New API", or NAPI for short. NAPI allows the driver to temporarily poll the NIC for new packets actively, instead of waiting for an interrupt, during times of high traffic[35]. This reduced the amount of interrupts generated, and thus reduced the overhead. However, to enable the use of NAPI, the driver has to support it.

Whenever a new packet arrives, the CPU will need to process it. If the CPU is not available, it needs to be stored until the CPU is ready. This is done by using buffers, implemented as ring buffers in the system memory. The NIC can copy packets into the buffer without using the kernel or CPU by using Direct Memory Access (DMA), allowing packets to be copied in the background. These buffers can then be used as a queue by the CPU. Queue sizes are set by the driver, and can in most cases be configured by the user. In addition to varying queue sizes, the number of queues can also be changed. This can be useful in multicore systems, where each core can handle one queue[30].

Once the CPU is ready to process a packet from the receiving queues, parsing can begin. At this stage, the incoming packet is parsed layer by layer until all headers have been processed. The kernel handles parsing of headers at the data link layer up to and including the transport layer. Checksums are also verified at this stage. The kernel will then copy the data contained in the packet to the receiving socket of the application. The application can then read the data and use it as needed. Note that a copy takes place here, which does have a performance cost. The copy operation is needed because the network stack runs in kernel space, while the application runs in

user space. This separation is made for security reasons, but does have a performance impact. A way to avoid this extra copy is described in subsection 2.2.4.

2.2.2 Tuning the network stack

The Linux network stack was designed to work for any application with minimal configuration. As a result of this, the performance may not be optimal for all applications. To improve the performance for a particular application, the network stack can be tuned to optimize relevant network metrics. For example, for a real-time application, it may be more important to reduce latency and jitter, even if it means lowering the maximum throughput or increasing CPU usage.

ethtool

One way of tuning the network stack, is to use the `ethtool` utility. `ethtool` is a command-line utility for changing parameters in the NIC driver, and by extension how the NIC operates. The parameters that can be changed, depend on what the driver reports as being configurable, but may include parameters such as queue sizes, queue counts and distribution, and interrupt settings. It may also be used to obtain information and statistics from a NIC, such as firmware version, received and transmitted packets, packet loss, and current packet rates[17, 30].

Threaded NAPI pool patch

As mentioned in subsection 2.2.1, the NAPI system allows the driver to poll the NIC for new packets, instead of waiting for an interrupt. However, where this polling and subsequent processing takes place, is not specified. In the default implementation, the polling and processing takes place anywhere on the system, making it more difficult to monitor and control[9].

In 2020, a patchset was published that would allow the polling and processing to be done in dedicated kernel threads instead of in software interrupt handlers. This would allow for better control of the system, and easier monitoring of resource usage for the NAPI. The patchset was merged in 2021, and can be used on newer kernels by setting a flag in the Linux system configuration. Initial tests showed some performance improvements, but the patchset is still very new, and not much data is available[9].

2.2.3 Alternative kernels

There are several alternative patchsets that can be applied to the Linux kernel, in order to fit specific use cases better. One of these patchsets, is known as the `PREEMPT_RT` kernel. The main feature of this kernel, is to allow for preemption.

This means that tasks can be assigned a priority, and tasks with higher priority can interrupt tasks with lower priority if required. This also applies to kernel tasks, where critical applications may take priority and preempt the kernel. These features can be useful for real-time applications, where applications might have strict timing requirements. Additional features of the `PREEMPT_RT` kernel include handling interrupts as kernel threads instead of directly in the interrupt context, and improve the accuracy of the kernel timers[1]. Together, the features of the `PREEMPT_RT` kernel pathset can help make a system more deterministic, and thus more suitable for real-time applications[18].

2.2.4 Bypassing the network stack

In some cases, such as when achieving maximum performance is required, it may be necessary to bypass the network stack. This can be done using *memory mapping frameworks*, such as Intel Data Plane Development Kit (DPDK) or `PF_RING ZC`. These frameworks allow the application to directly control the NIC, completely bypassing the kernel. This has two main advantages: allowing applications to use all the features available on the NIC, and potentially avoid the performance overhead of the network stack. This may give large performance improvements, at the cost of the portability and usability of the network stack[20].

The term *memory mapping framework* describes one of the core features of the frameworks: allowing the NIC to write packets directly into application memory, instead of writing it to the kernel. This allows the applications to access the packet data directly, instead of waiting for the kernel to process it. In addition to allowing full control over packet processing, this also avoids the need to copy packet data from kernel memory to user memory.

Using a memory mapping framework does have some drawbacks. One of them is that while the Linux network stack is designed to be general and support many different NICs, memory mapping frameworks are often tailored to support specific NICs. This allows the frameworks to utilize all hardware features available on the NIC, but requires applications to be written specifically for the framework and NIC.

In addition to losing the portability of the network stack, the application will also need to handle all the tasks that the network stack normally handles. For example, DPDK, as the name implies, only implements up to the data plane of the network stack. Applications will therefore in most cases be required to either implement the transport layer protocols themselves, or use additional tools like F-Stack[7] to do so.

DPDK

DPDK is an open source memory mapping framework developed by Intel. It is designed to allow high-performance packet processing by controlling the NICs directly. DPDK includes a set of libraries and drivers for a wide range of NICs and use cases. The source code is available under an open source license at GitHub[11]. The project was started by Intel, but is now a part of the Linux Foundation[33].

DPDK is designed to be powerful and flexible, but this comes at the cost of usability. Compared to some of the alternatives discussed above, DPDK will in most cases require more work to set up, learn, and use[20]. However, once this entry barrier is overcome, DPDK can vastly improve the performance of network applications[26, 15].

Other memory mapping frameworks

DPDK is not the only framework available to developers who want to improve the performance of their network applications. There are several other frameworks available, such as `PF_RING ZC`[28] and `netmap`[27].

`PF_RING ZC` ("Zero Copy") is a framework developed by ntop. Like DPDK, it uses custom drivers to control the NICs directly, without going through the kernel. It is designed to be easy to use, by giving developers access to a Python library to integrate into their own applications. It also includes a packet capture application, which can be used to capture packets from the network and write them to a file. The main downside of `PF_RING ZC` is that it is not open source, and requires a license to use. It is therefore not an option for our thesis.

`netmap` is a framework developed by Luigio Rizzo. It differs from DPDK and `PF_RING ZC` in that it is not provided as one or more libraries, but rather a modified version of the standard network system calls used by the Linux network stack. This can make it easier to integrate `netmap` into existing applications, but also means that it is not as powerful as the other frameworks.

In 2015, Gallenmüller et al.[20] compared the performance of DPDK, `PF_RING`, and `netmap`. They found that DPDK had the overall best performance, while `PF_RING` came in at a close second. `netmap` had the worst performance of the three, but was still significantly better than the Linux network stack for their test applications.

2.3 Cisco TRex

In our thesis, we will not use Intel DPDK or any other memory mapping frameworks directly. Instead, we will use software that builds on top of DPDK to improve the

performance of our testbed. We will use the packet generator Cisco TRex, which we will introduce in this section.

TRex is an open source traffic generator developed by Cisco. It utilizes the DPDK framework to generate large amounts of network traffic using commodity hardware. It is designed to be used in performance testing of network devices, and can generate tens of gigabits of traffic on a modern CPU. The traffic generator supports a wide range of protocols and applications, making it suitable for many different use cases. It is also highly configurable, allowing users to customize the traffic to their needs[5].

2.3.1 Modes of operation

Cisco TRex has three modes of operation: Stateless Mode (STL), Stateful Mode (STF), and Advanced Stateful Mode (ASTF). In STL mode, all packets are generated based on templates or user-defined scripts, and the generator does not keep track of any state. This makes it suitable for testing stateless devices, such as routers and switches. STL mode sends one type of packet, but the packet can be fully customized using the Scapy library for Python[36]. Stateless mode can be used from a console environment, controlled via a Python API, or through a graphical user interface.

The Stateful mode is the default mode of operation for TRex. In this mode, the generator replays a Packet Capture (PCAP) file containing real traffic. The addresses are replaced by TRex to multiply the traffic as if it originated from different devices. This mode is suitable for testing stateful devices, such as load balancers and firewalls. This mode is usable directly from the TRex Command Line Interface (CLI), by providing a configuration file with an IP address range and other parameters. In this mode, it is also possible to do latency measurements in parallel to the traffic. This is done by sending a special packet at regular intervals, and measuring the time it takes to receive a response.

The latency measurements in stateful mode uses hardware features of the NIC if available, or a software implementation if not. This means that the accuracy of the latency measurements can vary between different NICs. The latency measurements are also limited to running on a single CPU core, which can be a bottleneck for high-speed links. The idea is to run the latency checks as a separate, smaller stream of packets, while transmitting heavier traffic in parallel, thus providing measurements for only a sample of the packets. It is also not possible to change the packet size of the latency-measuring packets.

The last mode TRex supports, is the Advanced Stateful mode. This is an extended version of the stateful mode, with support for traffic where some headers may change. An example of this, could be a web server proxy that changes addresses in the packet, while preserving most of the original packet. This mode is implemented by including

a full TCP implementation, based on the FreeBSD network stack. This allows TRex to understand and analyze full TCP traffic. In the future, this mode will support more detailed latency measurements, but as of May 2023, this has not been properly implemented yet.

2.3.2 TRex alternatives

Cisco TRex is not the only open source traffic generator available. There are several other alternatives, such as *Ostinato*[29], *Ixia-C*[21], and *MoonGen*[13]. These three alternatives all build on top of DPDK, but they differ in other ways. We will briefly introduce these alternatives in this section.

Ostinato is an open source traffic generator, which describes itself as the "reverse of Wireshark". It has a graphical user interface, and supports replaying PCAP files as well as generating traffic based on templates. It works on both Linux, Windows, and macOS. On Linux, it is possible to utilize DPDK with the help of a plugin. This allows *Ostinato* to generate traffic at line rate. However, even if the source code of *Ostinato* is open source, the DPDK plugin is not. For most users, it is also necessary to purchase a license to use the software. This makes it unsuitable for our goals.

Ixia-C is another open source traffic generator. It is marketed as an open source alternative to the commercial traffic generator family *Ixia* from Keysight Technologies. *Ixia-C* is built as a reference implementation of the Open Traffic Generator API, which is a standard by the Open Traffic Generator organization. *Ixia-C* is published under a permissive MIT-style license. However, generating large amounts of traffic requires the commercial version of *Ixia-C*, known as *Keysight Elastic Network Generator*. Since other alternatives without this restriction are available, we will not use *Ixia-C* in our thesis.

The last TRex alternative we will present, is *MoonGen*. *MoonGen* is a scriptable high-performance packet generator built on top of DPDK. It uses Lua as a scripting language, making *MoonGen* very flexible. For common use cases, convenience scripts are provided. *MoonGen* is published under a permissive MIT license. While TRex was developed by Cisco as a tool for engineers, *MoonGen* was developed by researchers as an academic tool.

Among the presented alternatives, *MoonGen* looks like the most promising alternative. Both are actively developed and maintained, and support a wide range of NICs and systems. However, *MoonGen* is not as mature as TRex, and requires hardware support to do accurate packet timestamping. At the same time, the feature set and usability of *MoonGen* looks promising, so further research could be done to compare the two alternatives. For our thesis, we will use TRex.

Chapter 3
Testbed Creation

In the first part of our thesis, we designed and implemented a testbed solution for network devices, based only on open source software and COTS hardware. The testbed was not only designed to benchmark the Linux kernel, like we did in the second part of the thesis, but also to be general and reusable for other research projects in the future.

The testbed is based on Cisco TRex, with slight modifications to the source code to export more finely granulated data for processing.

3.1 Goals of testbed

Before we started designing a testbed, we needed to define what we wanted to achieve with it. We had to decide on what kind of tests we wanted to run, and what kind of data we wanted to obtain.

Since we were interested in measuring suitability for real-time applications, we decided that maximum throughput was not the most important metric to measure. Instead, we wanted to focus on the latency, especially on variations in latency. Existing solutions, like `iperf3`, give good average values, but do not show small outliers that can happen over time. We wanted to make an attempt at measuring these outliers, and see if we could find any patterns in them.

To achieve this, we needed a few things in the testbed:

1. A packet generator that could efficiently generate a large amount of packets
2. A way to store individual timestamps and latency values for each generated packet
3. A metric of analysis that gave insight into the anomalies in the data without being overwhelming

The first two points are closely related, since the packet generator will have to do the recording and storage of timestamps on the artificial traffic. The third point was more about how we wanted to present the data to the user, and how it would be processed, so it was mostly independent of the traffic generator. We wanted to do as little processing as possible while running the tests, to avoid affecting the results.

3.2 Hardware setup

Like in most of the tests in RFC2544, we wanted to test the performance of a network device in an isolated environment. This was to avoid any interference from other network devices, and to avoid affecting the rest of the local network. Like in the RFC, we will refer to the device we were testing as the DUT. The tester device will be referred to as the Packet Generator (PGEN). For this setup, we needed both the DUT and the PGEN to have two available network interfaces. This was to allow us to connect the two devices directly to each other, but still have data use different paths in each direction.

While the testbed is designed to be general and reusable, we will also give a description of the exact hardware setup used in this thesis. A photo of the testbed setup can be seen in Figure 3.1. The PGEN in the testbed used in the thesis was a Dell Precision 3640 Tower, which was inherited from a previous project at the department. It had the following technical specifications:

- CPU: Intel Xeon W-1270P, 8 cores running at 5.10 GHz
- Random Access Memory: 32 GB DDR4
- NIC: Intel Ethernet Network Adapter X710-T2L, firmware version 8.10 0x800093ea 1.2829.0
- Storage: 512 GB M.2 NVMe Solid State Drive (SSD)

The DUT in our testbed was mostly equal to the PGEN, except for having 64 GB of RAM instead of 32 GB.

Both machines used a fresh installation of Arch Linux from December 2022. The main reason for picking Arch Linux, was the availability of recent software packages, and good documentation. A different distribution, like Ubuntu or CentOS, would probably have worked just as well, since we could use the same software tools. The PGEN used kernel version 6.0.5.14.realtime1-2-rt, while the DUT used kernel versions 6.0.10.arch2-1 or 6.0.5.14.realtime1-2-rt depending on chosen testing configuration. The machines were connected using two 50 cm CAT6 cables.

Figure 3.1: Photo of testbed setup, with the PGEN and DUT connected directly to each other

In addition to the dedicated Intel NIC, both machines also had a built-in NIC on the motherboard. This interface was used for internet and remote access over SSH, while the Intel NIC was used for the actual benchmarking traffic.

3.3 Software implementation

The software implementation of the testbed was split into two parts: a modified version of Cisco TRex to serve as a traffic generator, and a data processing tool, named the data-postprocessor. The TRex fork was responsible for generating packets and storing the timestamps, while the data-postprocessor was responsible for analyzing the data and generating plots. We will introduce the traffic generator and validation of it first, and then move on to how we designed and implemented the data-postprocessor.

3.3.1 Cisco TRex fork

TRex was chosen as the packet generator for the testbed, because it is relatively well known and widely used among open-source alternatives[31]. It is actively developed, with several new commits added to the GitHub repository during the months we were working on this thesis.

When running TRex in stateful mode with latency measurements enabled, it will generate a report at the end of the test, and print it to the standard output. This report contains information about the physical links, such as the Ethernet line rate, as well as obtained measurements. A table with statistics for each network interface is generated, which shows total bytes and packet transmitted, packet loss, and malformed packets. At the end, a table with data from the separate latency measurements is printed. This table contains the average latency values, a calculated jitter value, and a histogram of aggregated latency values. This is a good start, but not enough for our purposes.

While TRex is a powerful tool, it was not designed to be used to inspect per-packet variations within a flow. Thankfully, because of its open-source license, this was something we could change ourselves by creating a fork of the source code. Our fork is located at https://github.com/KHTangent/trex-core. We created the fork on the 20th of January 2023, so any changes to the official TRex repository since then have not been included in the thesis. The fork is, as of writing this, listed as being almost two thousand commits behind the master branch of TRex. This is misleading, as the commit history of the original repository has suffered from a bad merge, creating a lot of duplicate commits. The actual difference between the two repositories are only a few commits.

In summary, we made one change to the source code of TRex: added the export of intermediate data to the latency measurement mode in TRex. In practice, whenever TRex is executed, a file titled `timestamps-[date]-p0.data` is created in the current working directory. This file contains raw double-precision floating point number data of transmit and arrival times for each latency packet, which can be processed externally later. See section 3.5 for more information on how we use the generated raw data.

While modifying the source code, we had to be very careful to not introduce additional latency to the obtained results, and to avoid running out of memory. To test for this, we ran TRex in loopback mode. In loopback mode, the two interfaces on the PGEN are connected to each other, instead of through the DUT. A photo of this setup is shown in Figure 3.2. In this configuration, all performance variations are caused by the PGEN, like the OS, hardware, or TRex. We first did some tests with an unmodified version of TRex to see a baseline, and then repeated the same tests after our changes had been applied, so we could see how much they impacted the performance.

For our first attempt at implementing timestamp exports, we simply generated an empty file at the start of the test, and then appended the raw timestamp values to the end of the file as packets arrived. This worked, but it was very slow, and it

Figure 3.2: Photo of PGEN connected in loopback mode

greatly affected the results we were obtaining compared to the original TRex. This made sense, as doing a file write at every single packet has a lot of overhead.

Our second attempt, was to cache all obtained data in a linked list in memory, and only write it to the file at the end of the test. A linked list was used because they support constant-time insertion, while a regular vector would require resizing and moving of elements when it reached its full capacity. This approach was much faster, but put a hard limit to how many packets a test could contain before running out of system memory. To make matters worse, using a linked list instead of an array or vector meant that every packet data had to include a pointer to the next packet, limiting the amount of data we could generate even more.

Our solution was to use a fixed size array to cache the data, and to write it to the file in chunks. This provided a good balance between speed and memory usage, and did not noticeably reduce the performance of TRex compared to the original version. We could implement this with relative ease, because the latency measurement mode in TRex runs on a single core, so we did not need to be careful about thread safety. All code changes were performed in the `src/stateful_rx_core.cpp` and `src/stateful_rx_core.h` files of the original TRex code.

Unfortunately, this method of obtaining results is not perfect. As mentioned, it only uses the latency measurement mode in TRex, which has several limitations. First off, it has to do CPU processing on every packet, severely limiting the maximum throughput we can generate. Secondly, the mode does not support setting custom packet sizes, and we were unable to add this feature during our work on the thesis. This means that we can only generate small packets. It could be possible to implement the latency exports in one of the other modes supported by TRex, but this would seemingly require a lot more work, and we did not have the time to do this. Another downside, is that latencies are only stored when the packet is received, not when it is sent. This makes it impossible to know when a packet is lost. An estimation for packet loss can still be made by looking at the difference between the expected and actual number of packets received.

3.3.2 Testbed validation

Before we could start using the testbed for our experiments, we had to validate that it was working as intended. We wanted to make sure that the results we obtained were accurate, and that the testbed was not introducing any errors. We did this by running TRex in loopback mode, and inspecting the results for any anomalies.

While running TRex in loopback mode, we would expect latency to be very low, and with little variation. However, we did see a few outliers in our preliminary tests, which would affect our results if we did not account for them. One option would be to do several tests in loopback mode, and make an estimation for TRex-induced outliers, then subtract them from the obtained test results. This would not be very accurate, and we wanted to avoid it if possible. A better solution would be to reduce or completely eliminate all TRex-induced spikes.

A plot of a 10-minute test run in loopback mode before our changes, can be seen in Figure 3.3. The vertical lines in the plot represent one or more packets that were delayed by a significant amount of time. We will revisit this type of plot later, but for now only the amount of vertical lines matter.

We took several steps to minimize spikes. Our main assumption, was that the spikes were caused by other processes on the system, causing small delays in the execution of TRex. We therefore started by going through all processes running on the system, and disabling the ones we deemed unnecessary. Our main change here, was to disable Xorg, the graphical user interface, in favor of running the system in terminal-only mode. This had an impact on our results, but did not resolve the issue completely.

Our second change to the PGEN system, was to let TRex run on isolated CPU cores. This would ensure that no other processes would run on the same cores

as TRex, potentially reducing the number of spikes. This was accomplished by modifying the boot parameters of the Linux kernel, and isolating four CPU cores. We then used the `taskset` command to force TRex to run on the isolated cores. Like when disabling Xorg, this had a positive impact on our results, but we still had some spikes left.

Finally, we migrated the PGEN to use the `PREEMPT_RT` kernel. As mentioned in the background chapter, this is a modified version of the Linux kernel, which is optimized for situations where applications need to respond within set time limits. On Arch Linux, this was easily achieved by installing the `linux-rt` package, and rebooting the system. This had a great effect on the remaining performance issues we had, and by our tests the TRex-induced spikes were practically gone. Our PGEN was now ready for usage. A new plot of a 10-minute test run in loopback mode after our changes, can be seen in Figure 3.4. Note that there are still some delayed packets right after the start of the test, motivating us to discard the first second of each test to account for such warmup time.

While it is possible to tune various network-related parameters in the Linux kernel, such as with `ethtool`, this would not have had any effect on our results. This is because TRex bypasses the Linux kernel completely, and communicates directly with the NIC using DPDK. There might be some parameters in DPDK that could be tuned, but we did not investigate this further.

3.4 Metrics to use

We needed a way to give insight into the outliers in a test, without overwhelming the user with too much data. We considered several alternatives, but decided on using a metric with "anomalies". We defined an anomaly as "a set of n consecutive packets with a latency greater than t". This metric has several advantages. For example, the thresholds n and t can be adjusted based on requirements, and the generated output can be summarized well by providing aggregate data for all anomalies in a test run. The value of t could also be set dynamically based on the average latency of the test, to get an accurate result without having to calculate a reasonable value for t beforehand.

One disadvantage of this metric, was that it does not necessarily show if anomalies themselves happen in bursts or are spread evenly. To help with this, we wanted a way to create a plot of the anomalies as well, so that the user could see the anomalies on a timeline. This plot needed some thought, to try to capture as much useful info as possible without being confusing to the viewers. Implementation-wise, this also meant that we would need to use a very flexible plotting library, or in the worst case we would have to implement our own plotting on top of a general graphics library.

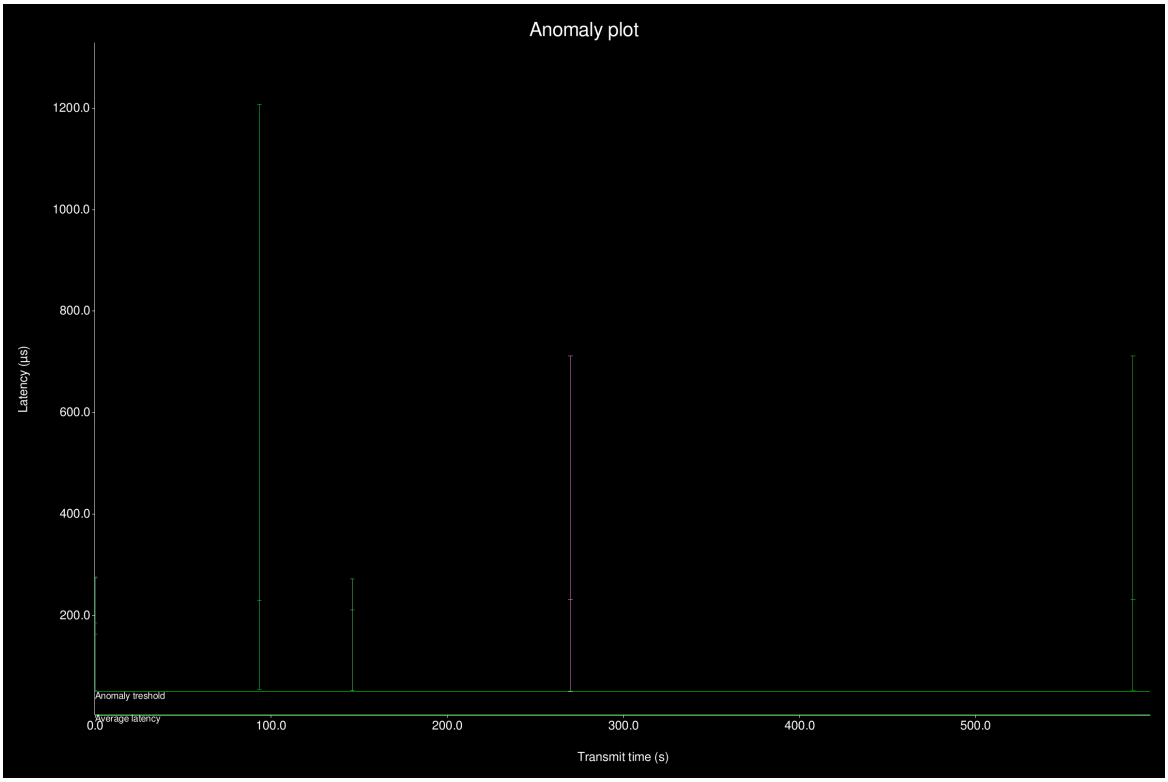

Figure 3.3: Anomaly plot of a test run in loopback mode with the default system configuration. $n = 1, t = 50\,\mu s$

Luckily, the plotting library we ended up using was flexible enough by itself, saving us time.

Before deciding on this metric, we also considered several alternatives. One alternative would be to create a plot of all latencies in a single second of data, and carefully inspect this. This would create a manageable amount of data for visual inspection, but would be difficult to do systematically. It would also not catch periodic events that happen less than once a second.

Another alternative was to make a plot of inter-arrival times. This would have shown variations in latency in a visual way, but would, like the alternative above, be difficult to do systematically. Nevertheless, we made an implementation of this in our data-postprocessor, should it be useful in the future.

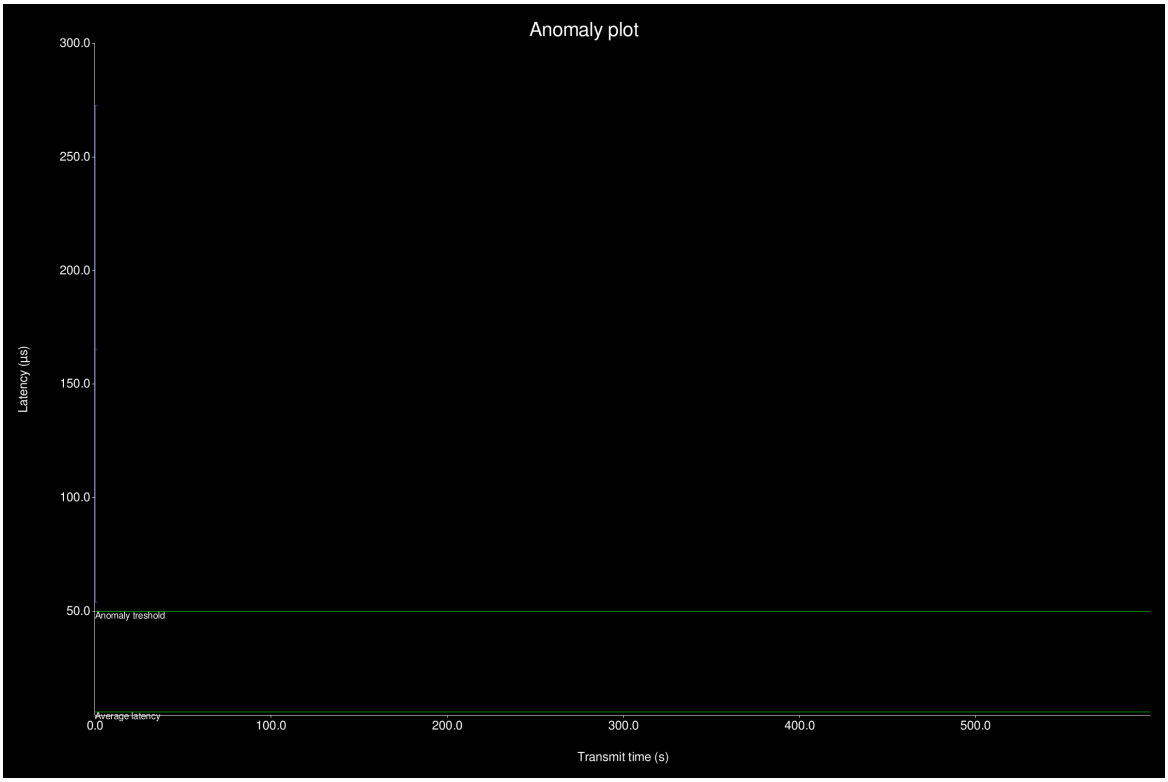

Figure 3.4: Anomaly plot of a test run in loopback mode with the optimized system configuration. $n = 1, t = 50\,\mu s$

3.5 Data processing

Obtaining data is not useful by itself, if there is no way to interpret and understand it. In this section, we will describe how we processed the data obtained from the testbed. First, we will describe how we wanted the data to be presented and what metrics to include, then we give a description on how we wrote software to parse the large data files generated by our testbed.

3.5.1 Processing large amounts of data

For maximum flexibility and minimum performance impact, our testbed only outputs raw timestamps for transmission and arrival of packets as double-precision floating point numbers. To properly utilize the testbed and obtain results, we would need to extract useful data and measurements from this. A simple way to plot the data, would be to just add a pixel to the plot for each latency measurement. This would give a decent general overview of the data, but would not be very useful for systematic

analysis, and would be overwhelming for large data sets. However, it was useful as a first step, and we therefore used it when evaluating different tools for processing and plotting.

Some of our tests generate large amounts of data. For example, a 10-minute test with 1 million packets per second would generate 600 million data points, or about 8.9 GiB of data[1]. This is a lot to process, making performance and memory usage more important than usual.

To be able to get some insight into the general "shape" of the data, we wanted to make a plot of all the data from a single test. This would require plotting a point for each data packet. We first tried to use the `matplotlib` library for Python, as we were already somewhat familiar with it. However, it quickly became clear that this would not work well. We used it to plot a file with 120 million data points, which took about two minutes. This does not sound bad by itself, but we knew that we would plot many more data files, and potentially large ones.

We made an attempt of using the Julia language with the `Plots.jl` library instead. Our results showed a slight improvement in performance, of about 40%. Even if this is good, it was still not enough to make plotting all our data feasible in a practical amount of time. Additionally, we did not have much experience with the Julia language, and it was not worth the effort to learn it well for such a small improvement.

We ended up rewriting our plotting script as an application in the Rust programming language, using the `Plotters` library[32]. This was a bit more work than using Python, but in the end we could plot our test file in two seconds instead of two minutes. This was a remarkable improvement, making it possible to plot all our data in a reasonable amount of time. At these speeds, the plotting time is not the limiting factor, but rather the time it takes to read the data from disk.

Writing a plotting script in Rust was also a good learning experience, as we did not have much Rust experience. Rust is a language that is designed to be fast and memory safe, but the design makes it a bit different from many other languages. For example, Rust does not have a garbage collector for memory management, but instead uses a system of ownership and borrowing. This means that the compiler can guarantee that memory is always freed when it is no longer needed, and that there are no dangling pointers, without the performance penalty of a garbage collector system[10]. This is a very powerful feature, but it also means that the programmer has to think about memory management more than in higher-lever languages. For

[1] 8 bytes per timestamp, 2 timestamps per packet (transmission and arrival time), 1 million packets per second, 600 seconds.

example, if a value is produced by a function, and then passed into a different function, it is considered "moved" and can no longer be used by the first function.

When writing the Rust application, we first implemented it by reading the entire data file into memory, then plotting it. This worked well for files up to a few gigabytes in size, but we wanted to be able to process larger files eventually. We therefore had to implement a buffered iterator reader, which would read the file in small chucks, while providing the rest of the application with a simple iterator interface. This was more work compared to caching the whole file in memory, but it was worth it in the end, as we could now process files of any size without running out of memory. Our tests showed that this implementation was able to handle files of up to at least 250 GB without problems.

To improve the user-friendlyness of the application, we also implemented a simple command-line interface, using the `clap` library[6]. `clap` is a library for parsing and validating command-line arguments, in addition to generating help messages and other useful features. This made it very easy to implement a simple interface for our application, which would be useful for future use.

All the plotting tools are available in the `thesis-subprojects` GitHub repository, located at https://github.com/KHTangent/thesis-subprojects. The original, unfinished Python and Julia scripts are located in the `scripts` directory as `latencyplotter.py` and `latenclyplotter.jl`, while the more polished Rust application is located in the `data-postprocessor` directory. Only the Rust application supports the detection and analysis of anomalies, the other scripts are only used for plotting raw latency values, and are preserved mostly for comparison purposes.

3.5.2 The data-postprocessor application

The finished Rust application, the `data-postprocessor`, has two main modes: plot mode, and validation mode. Both modes accept a single file as input, and will process it according to the rest of the given command-line parameters. The plot mode will plot data about individual packets, with an option for plotting individual latencies, or inter-arrival times. The plotting mode is limited to files that can fit in the system memory, since the plotting library requires all data points to be in memory while plotting. This mode can be useful to get a quick overview of how a data file looks, but it is not very useful for systematic analysis.

The other mode, validation mode, is more useful for systematic analysis. This mode will go through the data file, and look for packets that are part of a latency spike, which is how we defined an "anomaly" earlier. Thresholds for what is considered an anomaly can be set with command-line parameters. The application will then print out some statistics about the data (such as average latency), and a list of all

anomalies found in the file. Optionally, it can also generate a plot of the anomalies, to give a graphical representation of when anomalies happened, and how severe they were.

The application is cross-platform, and can be compiled and run on any platform supported by the Rust compiler. This includes Windows, Linux, and macOS. This allows future users of the testbed to keep processing and analysis of the data on their own computers, without having to use the testbed itself. This can be useful if the testbed is not available, or is being used for other tests.

3.6 Documentation

To make it easier to use the testbed, we wrote a user guide. This guide describes how to set up the testbed, optimize the packet generation, how to run tests, and how to analyze the data. It also includes some examples of how to use the testbed, and how to interpret the results. The guide is available in our GitHub repository, at https://github.com/KHTangent/thesis-subprojects, in the `testbed-docs` directory. It is written as simple Markdown documents, which can be viewed directly on the GitHub website. It is split into multiple chapters, with an introduction chapter that explains the different parts. We have included a copy of the guide as an appendix, see Appendix A.

Now that we had a usable testbed, and a way to analyze the data, we were ready to put it to use and start running tests. We will describe the tests we ran in the next chapter.

Chapter 4
Linux Evaluation

Now that we had created our testbed, it was time to put it to use. We used the testbed to evaluate the performance of the Linux kernel network stack, with a focus on real-time applications. In this chapter, we will first describe the goals of the testing by introducing what we tested, then describe the methodology used for the test. Afterward, we will present and discuss the obtained results.

By doing these tests, we were hoping to find some results that could provide answers to some of our research questions. In particular, if we found some configurations that performed better than others, we could use this to answer **RQ3** in the context of real-time applications.

4.1 Variations to test

Before we could start the testing, we wanted to make a list of variations to try out. To make this list, we had to decide on some parameters we wanted to vary, and which values to try out for the parameters. Note that while we did various optimizations on the PGEN in the previous chapter, we will now do tests on the DUT, and all configuration parameters below are changed on the DUT.

RX/TX Queue Sizes The first parameter we wanted to vary was the size of the RX and TX queues. The RX and TX queues are used to buffer packets that are waiting to be processed by the kernel. The default size of these queues on our NIC was 512. This number represents the number of pointers to packet data structs, which in essence describes how many packets can be in queue before packets need to be dropped[37]. We wanted to test if changing this value would affect the results, and how. These values can be set using the following `ethtool` command:
`ethtool -G [interfacename] rx [size] tx [size]`

Kernel to use We had observed how the `PREEMPT_RT` kernel had reduced the latency variation in traffic generated by TRex on our PGEN, and therefore wanted to see how it would affect packet processing in general. To have a valid comparison, we tested with both the stock kernel, and with the `PREEMPT_RT` kernel.

Threaded NAPI pool patch As mentioned in chapter 2, it is now possible to let the polling of packets be handled in dedicated threads, which can be pinned to specific cores. We wanted to test if this would have any effect on the performance of the network stack, and if so, how much. We tested with and without enabling this patch.

The patch is enabled using the sysfs interface. To enable the patch, the following command was used:
`echo 1 | sudo tee /sys/class/net/<iface>/threaded`
After setting this option, NAPI-related processed started to appear in the process list. To set a priority for all these threads in one go, the following command was used:
`ps aux | grep '\[napi' | awk '\{ print $2 \}' | head -n -1 | xargs -I pid -n1 sudo taskset -pc 0-3 pid`
The command works by first listing all processed related to threaded NAPI handling (they were prefixed with `\[napi`), then extracts the process IDs using `awk`. Afterward, it removes the last line, which is the process that is running the `grep` command. Finally, it uses `xargs` to run `taskset` on each of the process IDs, and sets the priority of the processes to the first four cores. We had isolated these four cores using boot parameters, so they should not be occupied by any other processes.

System load When an application is running on a system, it usually does not only send and receive network traffic, but will also do some processing on the data. This puts some load on the CPU, which might affect the performance of the network stack. We wanted to test how the network stack would perform with minimal system load, and with a moderate amount of system load (40% CPU usage). To generate the system load, we used the `stress-ng` command, which is a tool for generating system load. We used the following command to generate the load:
`stress-ng -c 12 -l 40 -t 1y`
This creates 12 threads, one for each non-isolated core on our system, and sets the load to 40% for each thread. The load is set to run for 1 year, to make sure it does not stop during our tests.

Traffic amount Different applications send different amounts of traffic. We wanted to test how various amounts of traffic were handled. We used four different amounts of traffic: 1, 10, 100 and 900 Mbps. This is, unlike the other parameters, controlled by adjusting the traffic amount used on the PGEN. This gives four different magnitudes

of traffic. The reason for choosing 900 Mbps instead of 1 Gbps was that sending 1 Gbps of traffic seemed to cause some problems with the PGEN, where the CPU was not able to track all latency measurements in time. This caused packet loss in loopback mode, which was not acceptable for testing. We therefore chose to use 900 Mbps instead, which appeared to be stable.

Our TRex setup operated with packets per second, not bits per second. To convert between the two, we did some experimentation to determine what traffic amount gave the desired traffic rate. We found that 1900 packets per second gave a traffic rate of 1 Mbps, and used this as a base for the other traffic rates. The final traffic rates we used were therefore 1900, 19000, 190000 and 1700000 packets per second.

Test duration We wanted to test how the network stack performed over time. We therefore ran each test for a duration of 10 minutes. This should be enough to make sure the results are stable. While processing the results, we also stripped off one second at each end of the data, to account for warmup and cooldown time.

4.1.1 Final parameters

To summarize, these were the parameters we wanted to vary, and their values:

1. **RX/TX Queue Size**: We used three values: 256, 512 and 1024 packets, where 512 was the default value for our driver.

2. **Kernel to use**: We tried out both the stock kernel, and the `PREEMPT_RT` kernel.

3. **Threaded NAPI pool patch**: We tested with and without enabling a threaded NAPI pool.

4. **System load**: We used two different system CPU loads: 0% and 40%.

5. **Traffic amount**: We used values of 1900, 19000, 190000, and 1700000 packets per second, corresponding to 1, 10, 100 and 900 Mbps.

To give stability to the results, we ran each test 15 times, with each test lasting for 10 minutes. This gave a total of $3 \cdot 2 \cdot 2 \cdot 2 \cdot 4 \cdot 15 = 1440$ tests, lasting for a total of 10 days, plus time configuration time between each set of tests. In total, performing all the 10-minute tests took about two weeks.

4.1.2 Longer tests

In addition to doing the tests described above, we also did some longer tests to see how the configurations performed over longer periods of time. Due to time constraints, this was only done for the default settings, and for the setup that gave the best results in the shorter tests. These longer tests were done for 24 hours each, with a traffic rate of 100 Mbps (190000 pps).

4.2 Methodology

In the previous chapter, we described how we designed a testbed to evaluate the performance of a network device. We will now describe how we used this testbed to evaluate the performance of the Linux kernel network stack.

The general procedure for performing a test, was:

1. Power on both the PGEN and the DUT
2. Configure the DUT to forward packets, and apply the configuration that would be tested
3. Start TRex on the DUT to generate traffic for a set amount of time
4. After the test has finished, stop TRex and process the generated `.data` file with the data-postprocessor

Since we ended up with wanting to do 1440 tests, we needed a way to automate the testing procedure. Automating everything would have been a lot of work, so we decided to automate the execution of TRex, while doing configuration on the DUT manually. We wrote a simple script to run TRex several times with different traffic amounts, and naming the generated data files appropriately. This allowed us to run up to 60 tests in a row with no interaction (15 repetitions of 4 different traffic amounts). We then did the configuration on the DUT manually, and repeated until all 24 different DUT configurations had been tested. The batching script is, along with all other files used during the thesis, available on our `thesis-subprojects` GitHub repository, in the `batching` directory. Even if it is available, it is not by itself useful for others, but can serve as an example.

4.3 Data processing

Now that we have described how we collected the data, we will describe how we processed it. We used the `data-postprocessor` tool we developed for this purpose. Using a shell script, we ran the tool on all our 1440 data files, storing all visual plots

in separate folders based on configuration, and stored a summary of all runs in a text file. We then wrote some Python scripts to generate aggregated bar charts from the summary file. The scripts we made are available on our GitHub, but are not by themselves useful for others, as they are very specific to our data and naming schemes.

Our testbed was designed to accept user-defined values for the tresholds used to determine what is considered an anomaly. Both the parameters n for number of subsequent delayed packets, and t for the time threshold, could be set by the user. This meant that we had to decide on some values when using the testbed on the Linux kernel.

One possible source of information for deciding on the values to use, is RFC8578 [23]. This RFC talks about use cases for deterministic networking, and describes some requirements for the different use cases. One of the mentioned use cases is smart electric power grids. For this use case, the RFC gives a requirement of less than 750 µs delay variation for legacy traffic, and 250 µs for time-sensitive traffic. We can use these values as values for t in our metric. Even if our t describes an absolute latency, and not a variation, we can still use it in many cases. The baseline values for our DUT is very low, so we can approximate the variation as the absolute latency.

Finding a good value for n is not as simple. A simple approach would be to use $n = 1$, and count every delayed packet as an anomaly. However, the RFC mentions that some packet loss is acceptable, so we can assume that an application will be able to handle some out-of-order packets. We therefore used $n = 2$ as our value for n. This means that we will not count a single delayed packet as an anomaly, but two or more delayed packets will be counted as an anomaly.

The data-postprocessor tool also accepted a parameter to ignore the first and last x seconds of the input file, to account for warmup and cooldown time. We used a value of 1 second for this parameter, as our experiments showed that the results stabilized quickly.

4.4 Results

With 1440 tests performed, and several different values of n and t to consider, we ended up with a lot of data. It is not possible to give a thorough analysis of all the data in this thesis, so we will focus on aggregated values. We will also look more carefully at the two 24-hour tests. This section will only present results, and not discuss them. The discussion will be done in section 4.5.

For the rest of the section, we will refer to the different configurations with a systematic naming scheme. The naming scheme will use the following form:

`[kernel]-[threading]-[queuesize]-[idle/load]`
An example is:
`default-threaded-default-load`
This example refers to the stock (non-RT) kernel, with threaded NAPI enabled, default RX/TX queue sizes (512 packets), and a 40% CPU load. The traffic amount will be given separately. We have generated horizontal bar plots based on the obtained results over the 15 test runs. The plots show the sample mean of the 15 runs of each configuration, with error bars showing a 95% confidence interval for the results. In these plots, we have applied a separate color to each combination of kernel and threading, to make the figures more readable.

The 95% confidence interval is calculated by assuming that the obtained results are approximately normally distributed. We then calculate the average as an estimate for the mean of the distribution, and calculate a sample standard deviation as an estimate for the standard deviation of the distribution. We then calculate the 95% confidence interval by using the Student's t-distribution with $n-1$ degrees of freedom, and a confidence level of 95%. The t-distribution is used because we do not know the true standard deviation of the population, and have to estimate it from the sample standard deviation.

4.4.1 Latency results

We will start by looking at the aggregated results for the different configurations. A good start is to look at average latencies for the different configurations. We have plotted the average latencies for all configurations in figures 4.1, 4.2, 4.3 and 4.4. Note that for the 1700000 pps plot, we have used a logarithmic scale on the x-axis, while the other plots are linear. A plot of the packet loss for all configurations at 1700000 pps is also included, in Figure 4.5. The other traffic rates had no packet loss, so we did not plot them.

The first thing to note, is that most configurations had problems with packet loss at 1700 kpps. The only configurations that did not have any substantial packet loss, were the three different queue sizes for the default kernel without threaded NAPI. This is also shown in the latency values for this traffic rate, where the packets that made it through were delayed by a lot. These packet loss values are so high that we are not able to use the other results for this traffic rate, as they would be too skewed from the packet loss.

Another thing to note, is that results seem very stable. Most values have a narrow confidence interval, shown by the small error bars. This is good, and shows that 15 repetitions were enough to get a good result for the configurations.

An interesting takeaway from the results, is that the low-traffic run with 1900 pps

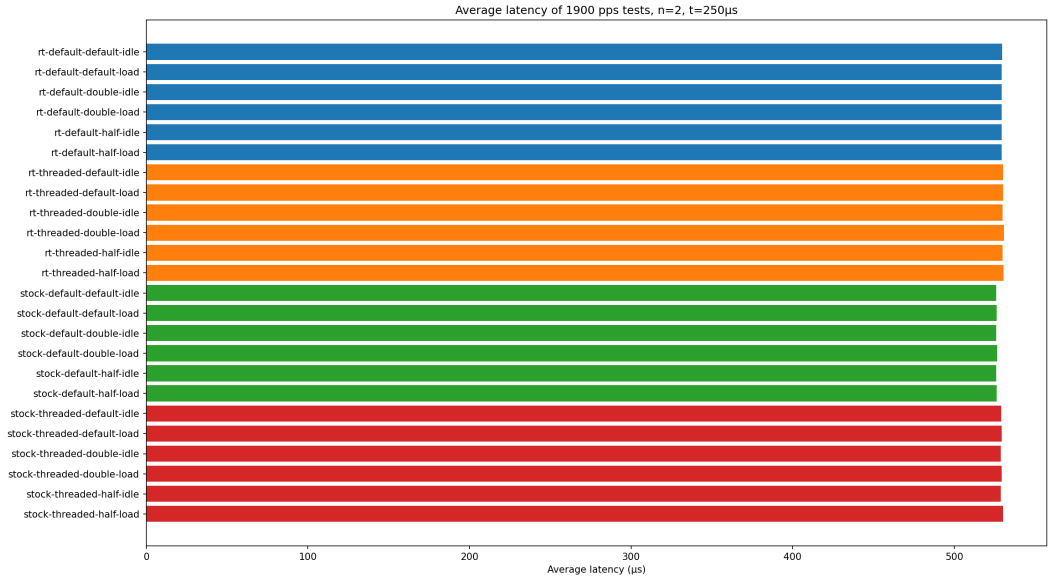

Figure 4.1: Latency for all configurations, at 1900 pps

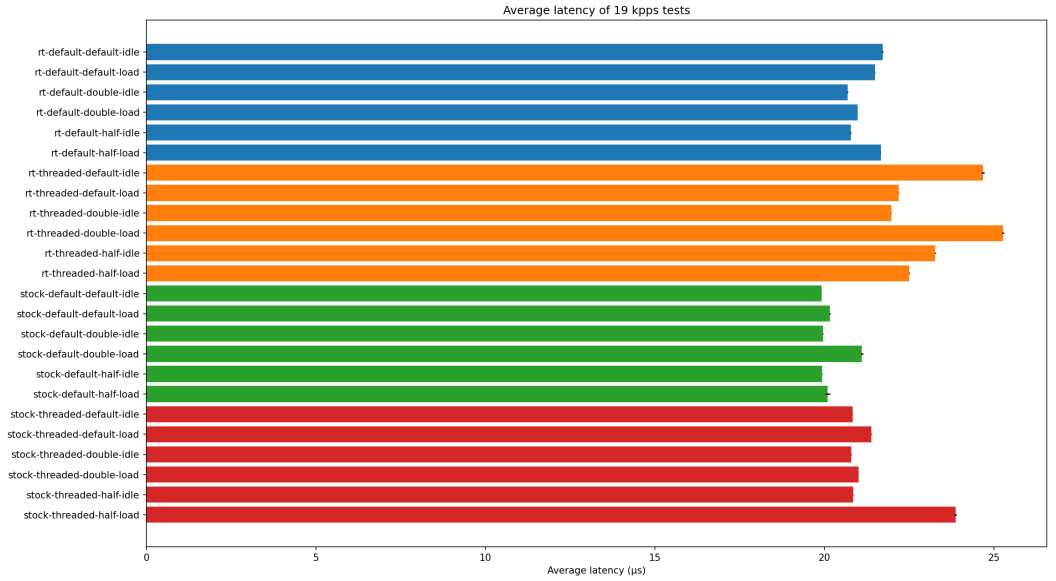

Figure 4.2: Latency for all configurations, at 19000 pps

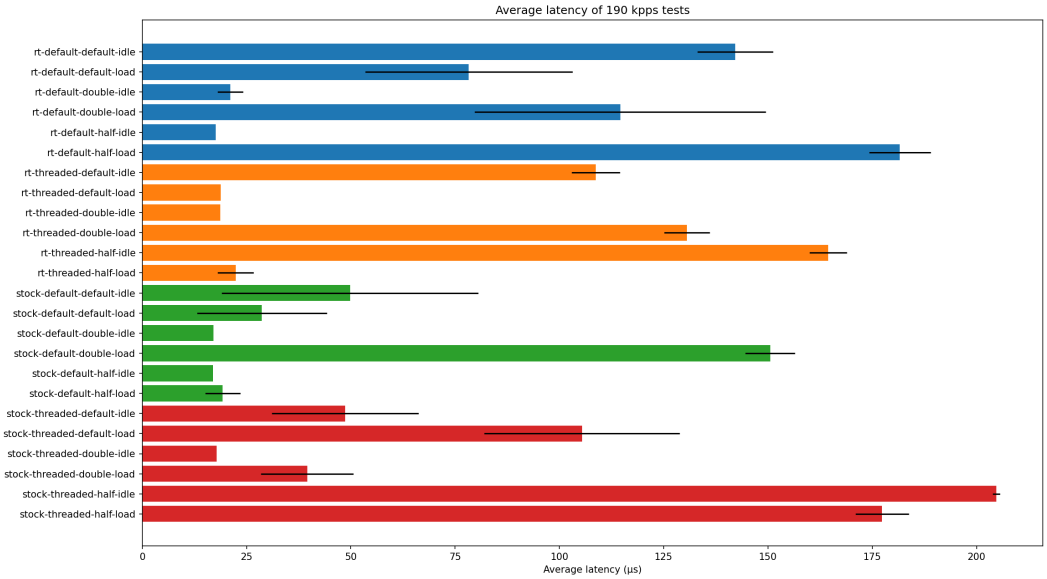

Figure 4.3: Latency for all configurations, at 190000 pps

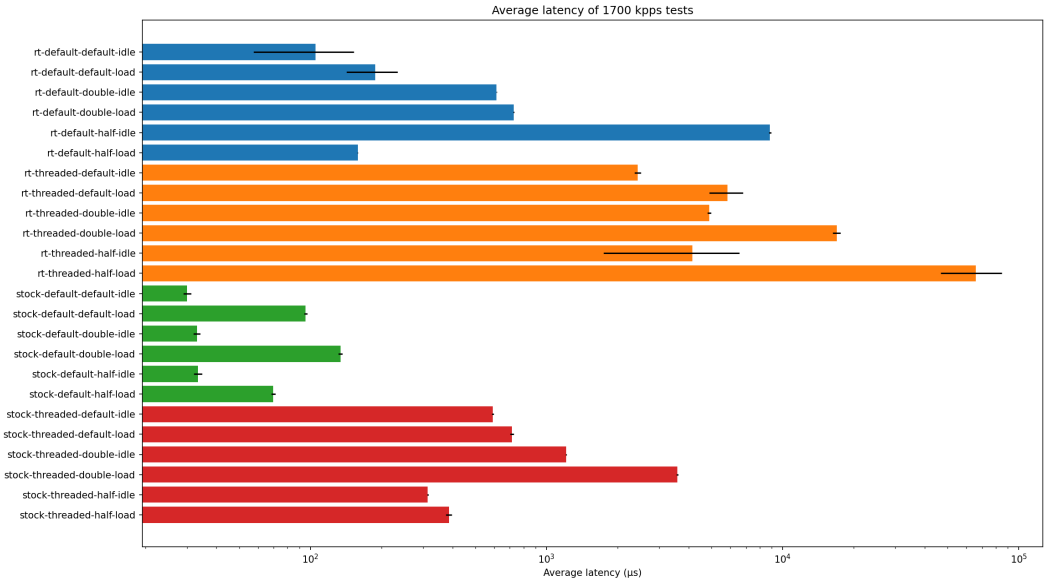

Figure 4.4: Latency for all configurations, at 1700000 pps

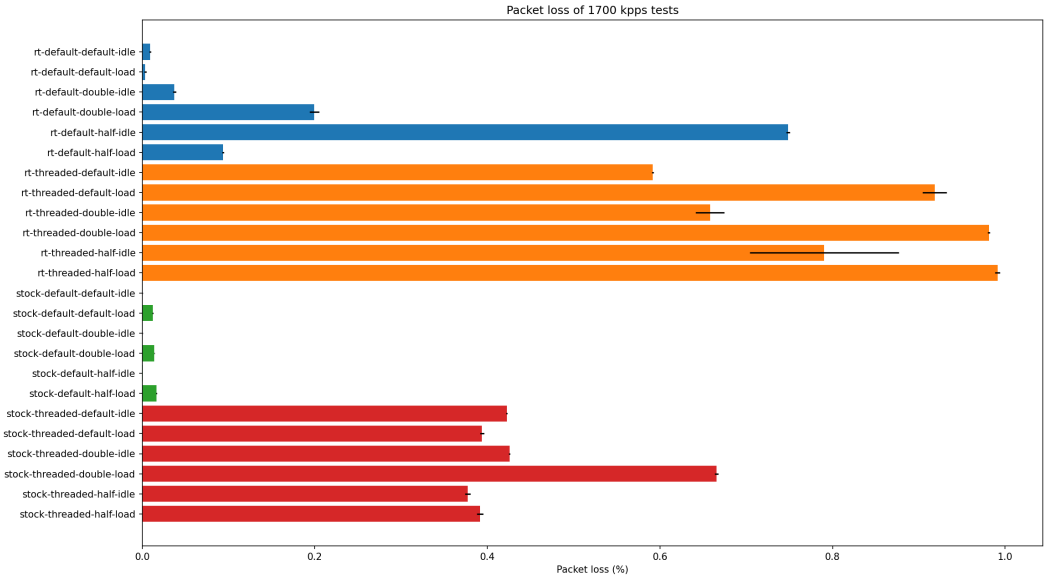

Figure 4.5: Packet loss for all configurations, at 1700000 pps

resulted in much higher latencies than the other configurations. The 19 kpps run had much lower latencies, and a bit more variation between the different configurations. The RT kernel seems to perform slightly worse than the stock kernel at this traffic rate, but the difference is not very large. The threaded NAPI configurations also seem to perform slightly worse than the non-threaded ones, but again, the difference is not very large. The queue sizes and system load values did affect the results, but does not show a clear pattern to draw any conclusions from.

The 190 kpps run shows more interesting results, with some configurations performing much worse than others. Some of the results are also very strange, for example that the `rt-threaded-half-load` configuration had an about 85% lower latency than the `rt-threaded-half-idle` configuration, but a larger variation.

4.4.2 Anomaly results

After looking at the latency results, we will now look at the anomaly results. Since we have two different values for n and t, this doubles the amount of data points compared to the latency measurements above. We have excluded the 1700000 pps traffic rate, as the packet loss was too high to get any meaningful results. We have

also excluded the 1900 pps measurements for $t = 250\,\mu s$, as the average latency in this test was much higher than the anomaly count. This would cause the result to appear as a single, long anomaly. The results for the other traffic with thresholds of $n = 2$ and $t = 250\,\mu s$ are shown in figures 4.6 and 4.7. Results with thresholds of $n = 2$ and $t = 750\,\mu s$ are shown in figures 4.8, 4.9, and 4.10. Note that the numbers along the x-axis change between the plots, as some configurations had many anomalies.

In addition to anomaly count, we will also look at the average anomaly duration. This is a measure of how many packets an anomaly lasted on average. Results with $n = 2$ and $t = 250\,\mu s$ are shown in figures 4.11 and 4.12. Results with $n = 2$ and $t = 750\,\mu s$ are shown in figures 4.13, 4.14, and 4.15. Note that data from runs with no anomalies have been excluded when calculating the average for a configuration. For configurations where there were no anomalies in any of the runs, the average anomaly duration has been set to zero.

The first thing to note, is that no configurations managed to get zero anomalies with the $t = 250\mu s$ threshold value. This is a bit disappointing, as we would have liked to see some configurations that "passed" the validation without anomalies. The results looked better for the $t = 750\,\mu s$ threshold, where some configurations managed to get zero anomalies.

As expected, more traffic resulted in more anomalies for all configurations with more than zero anomalies. This makes sense, as there are more packets that can be registered as anomalies.

Compared to the average latency tests, there are more patterns to be found in the anomaly counts. For example, configurations with no other system load and a doubled queue size perform well in almost all cases. In addition, the `rt-default-half-idle` configuration seems to perform well in all cases. For this reason, we decided to use this configuration in one of our 24-hour tests (see next subsection). Other parts of the result seem less predictable. For some tests, increasing CPU load actually improved the results, while for others it made it worse.

4.4.3 24 hour tests

The tests in the previous section were run for 10 minutes each. To get a better idea of how the system behaved over time, we also did two 24 hour tests. The first test was run on the default configuration with no system load, and the second test was run on a configuration using the real-time kernel, with halved queue sizes, and no system load. We used 190 kpps as data rate for both tests. The results from these tests were plotted using our data-postprocessor. Like with the 10-minute tests, we used two different threshold values, $t = 250\,\mu s$ and $t = 750\,\mu s$, both with $n = 2$.

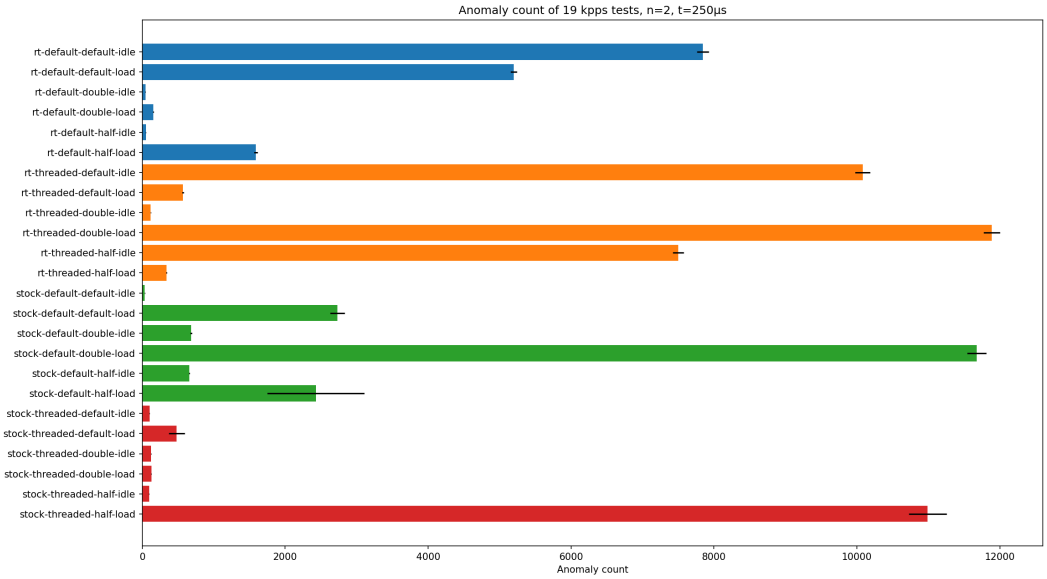

Figure 4.6: Anomaly count for all configurations, at 19000 pps. $n = 2$, $t = 250$

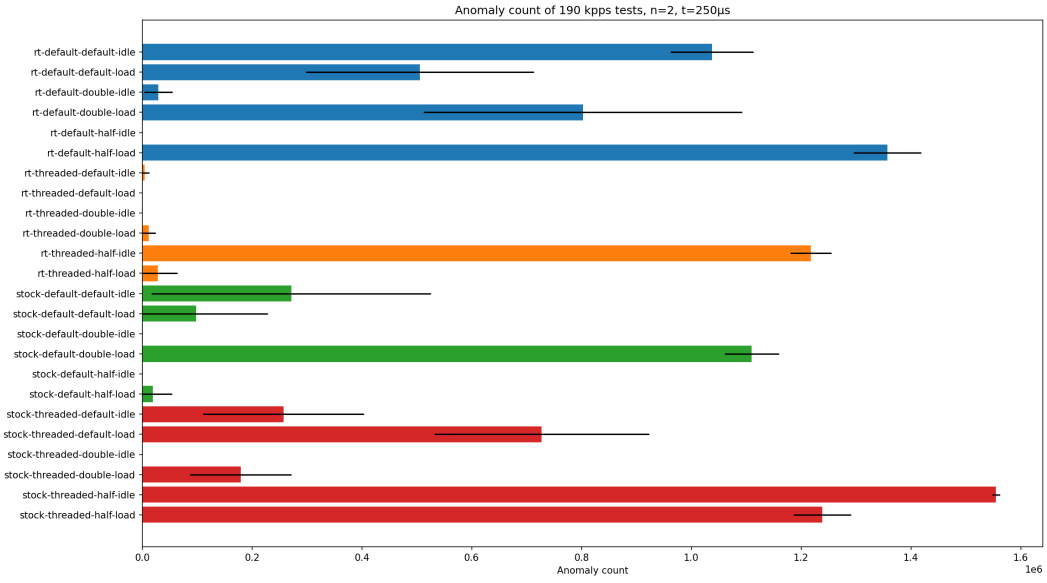

Figure 4.7: Anomaly count for all configurations, at 190000 pps. $n = 2$, $t = 250$

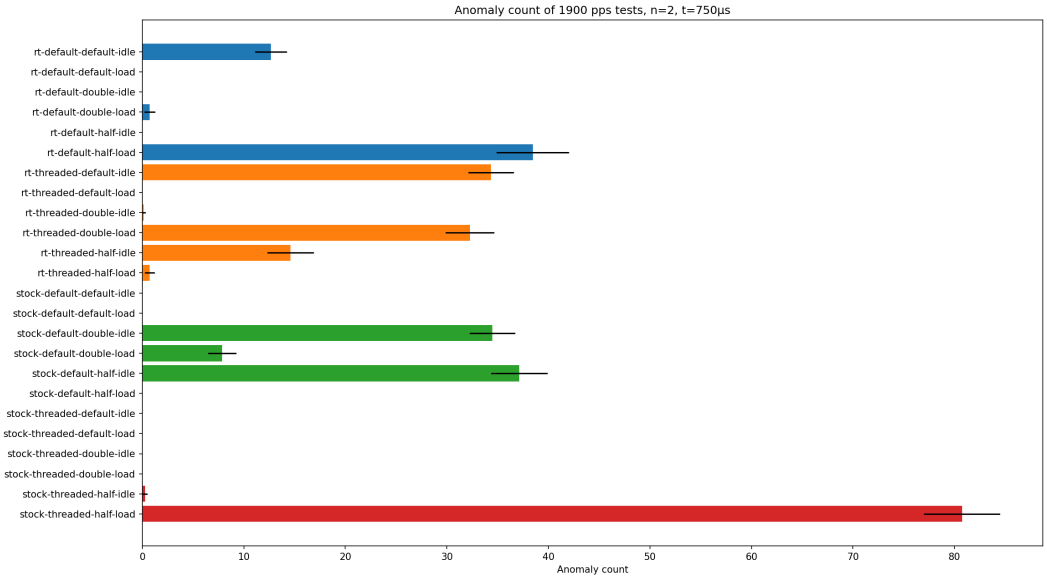

Figure 4.8: Anomaly count for all configurations, at 1900 pps. $n = 2$, $t = 750$

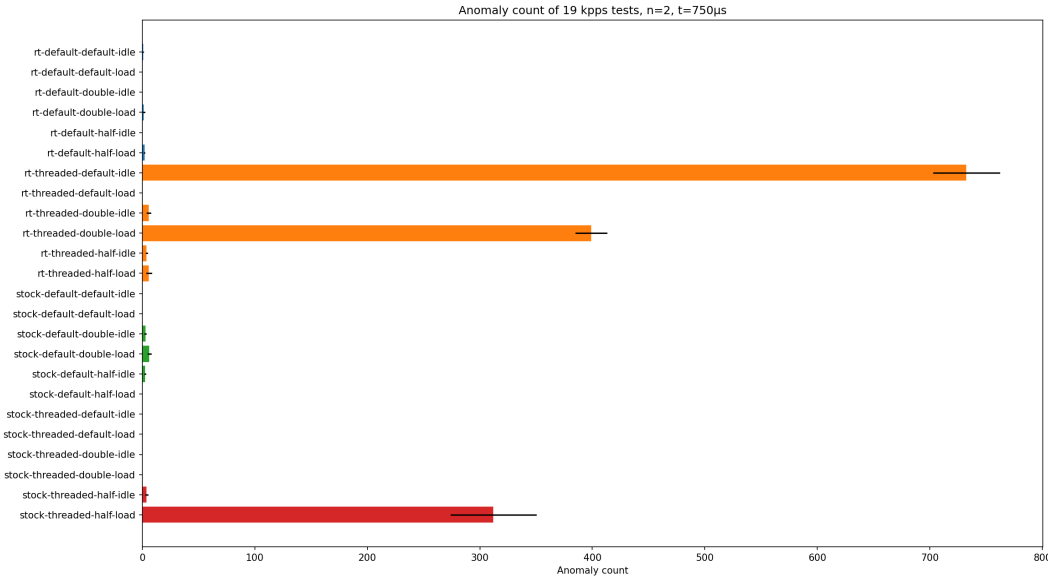

Figure 4.9: Anomaly count for all configurations, at 19000 pps. $n = 2$, $t = 750$

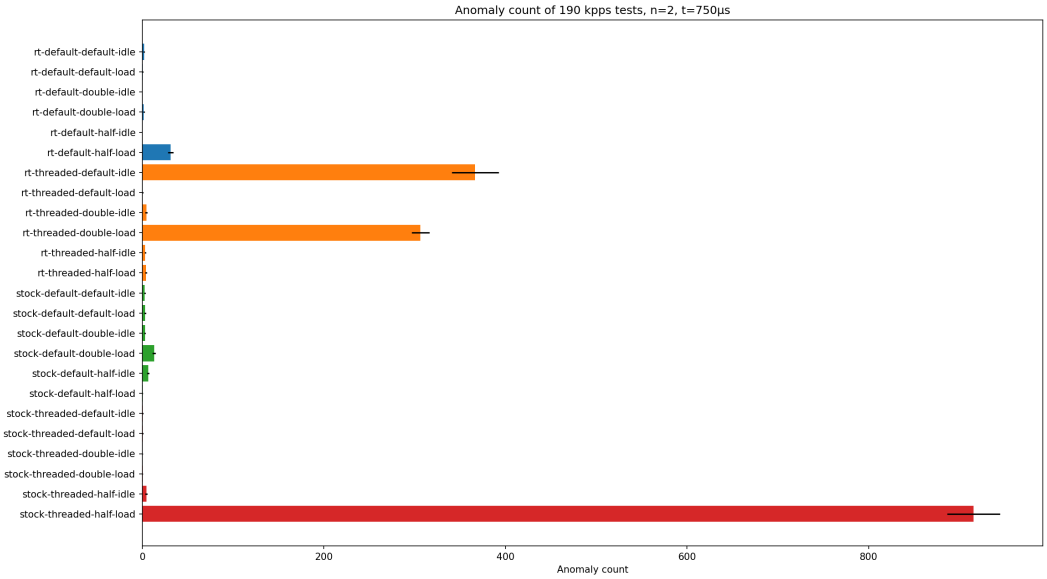

Figure 4.10: Anomaly count for all configurations, at 190000 pps. $n = 2$, $t = 750$

The data-postprocessor validation plots require some explanation, as they contain a lot of information. Each anomaly is represented by a vertical line, with three markers. The top marker represents the maximum latency within that anomaly, the bottom marker represents the minimum, and the final marker represents the average. The colors of the anomaly lines are arbitrary, and is only used to distinguish between different anomalies. The background color of the plots has been set to black, as this made it easier to distinguish between the colors of the anomalies. The duration of the anomaly is not included in the plot. A horizontal line is added to the plot to show the threshold value for t. There is also a horizontal line to show the total average latency, but this line is often not usable, since the average latency will be much lower than the threshold value, causing it to blend into the x-axis.

The results with the default configuration are shown in figures 4.16 and 4.17, while the results with half queue size and real-time kernel are shown in figures 4.18 and 4.19. Numerical results obtained from the data-postprocessor are shown in table 4.1.

The 24-hour tests give some insight into how the system behaves over time. There are some very severe anomalies that happen at most every few hours, and some

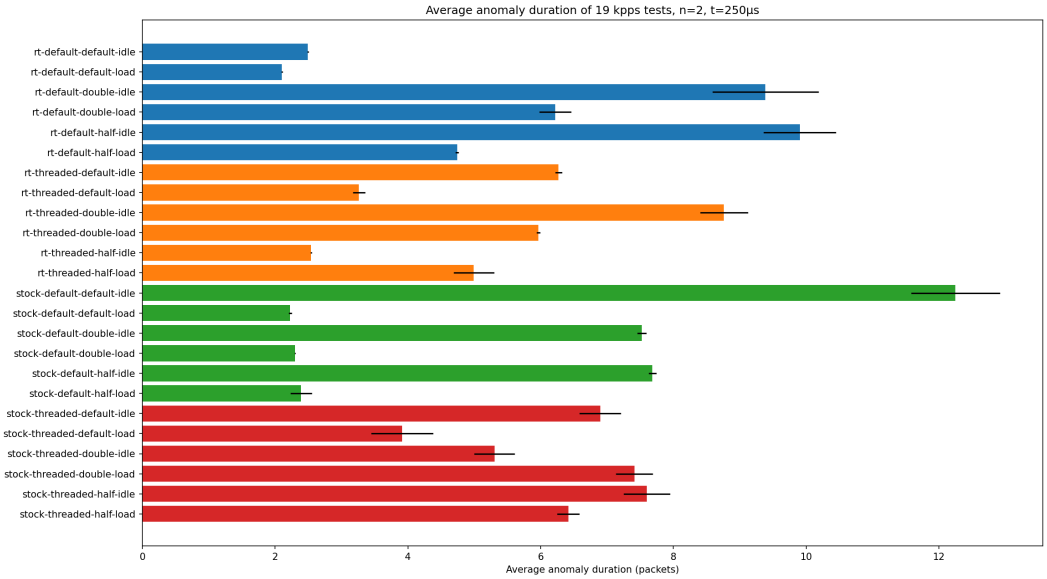

Figure 4.11: Average anomaly duration for all configurations, at 19000 pps. $n = 2$, $t = 250\,\mu s$

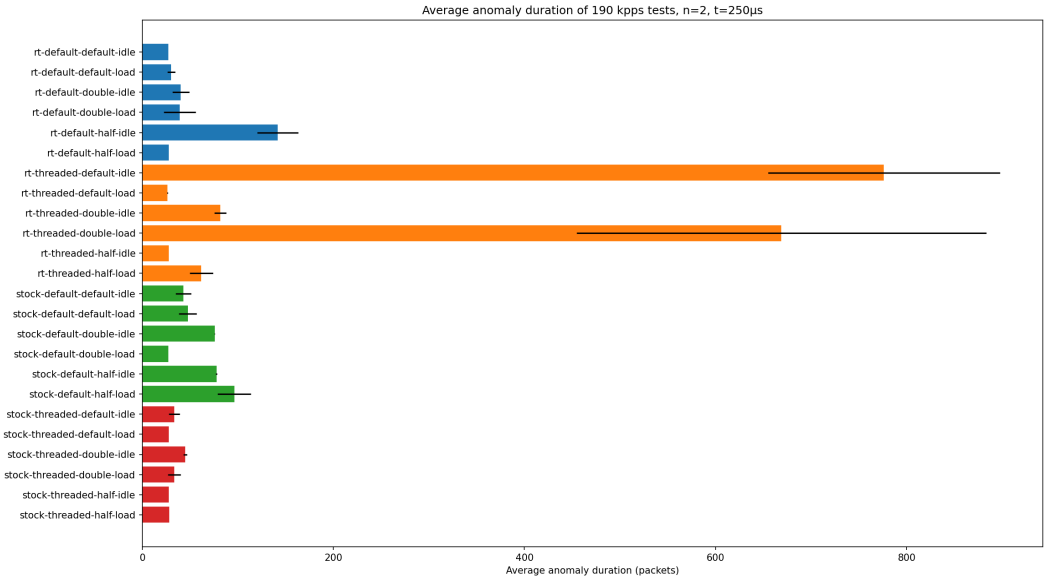

Figure 4.12: Average anomaly duration for all configurations, at 190000 pps. $n = 2$, $t = 250\,\mu s$

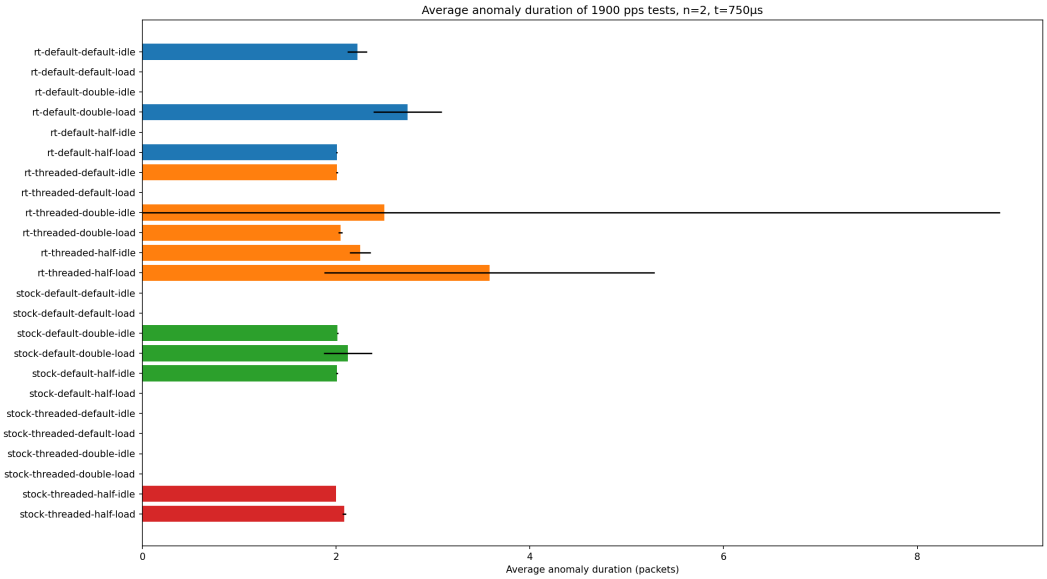

Figure 4.13: Average anomaly duration for all configurations, at 1900 pps. $n = 2$, $t = 750\,\mu s$

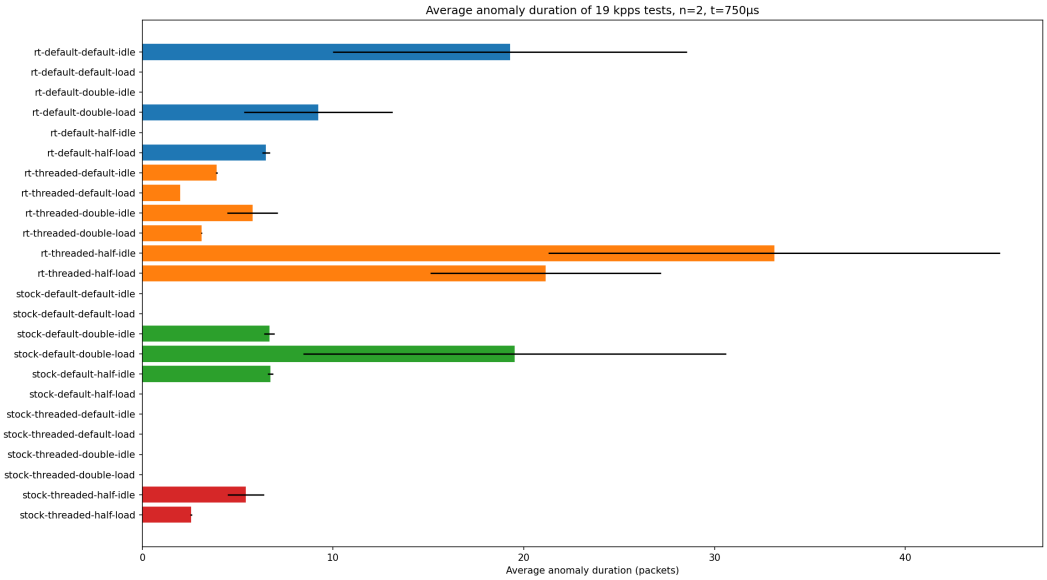

Figure 4.14: Average anomaly duration for all configurations, at 19000 pps. $n = 2$, $t = 750\,\mu s$

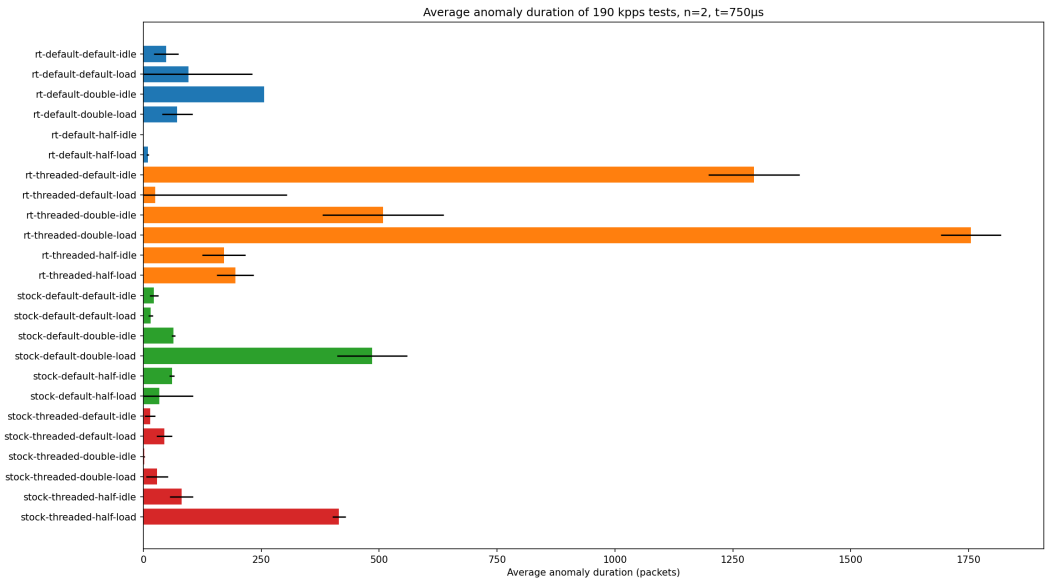

Figure 4.15: Average anomaly duration for all configurations, at 190000 pps. $n = 2$, $t = 750\,\mu s$

	Default $n = 2, t = 250\mu s$	Default $n = 2, t = 750\,\mu s$	rt-half-queue $n = 2, t = 250\,\mu s$	rt-half-queue $n = 2, t = 750\,\mu s$
Latency (min/avg/max)	10.790/20.923/8282.788 µs		11.081/18.075/4256.964 µs	
Anomalies	4101730	433	190	27
Average anomaly duration (packets)	27.471	76.820	172.621	40.704
Average anomaly packet latency	321.036 µs	991.845 µs	484.566 µs	1368.896 µs

Table 4.1: Numerical results for 24 hour tests

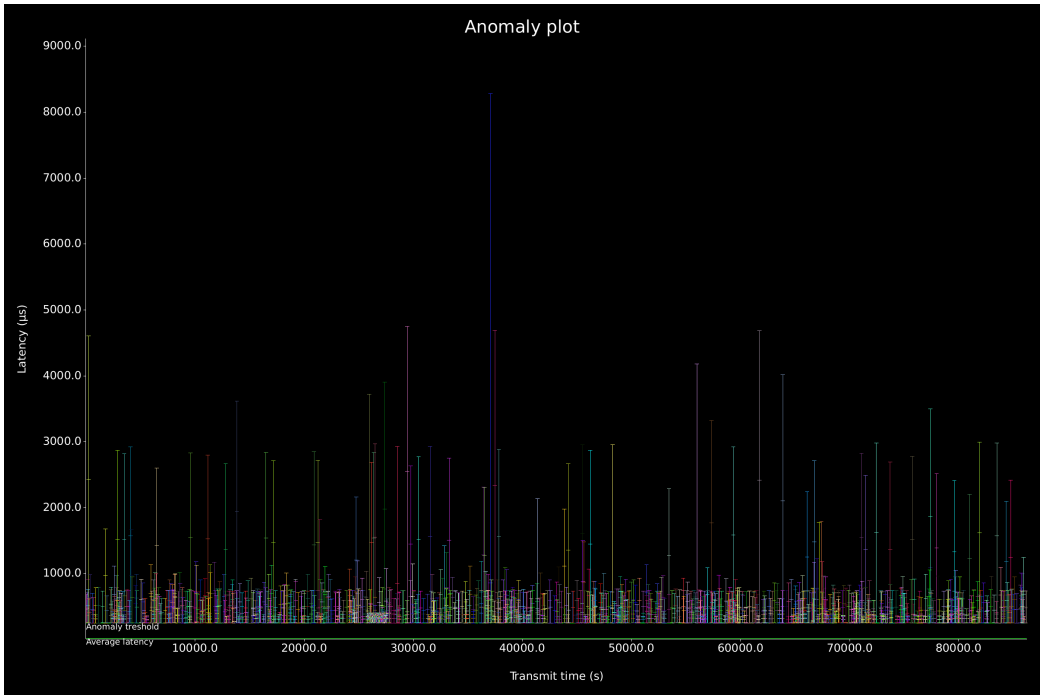

Figure 4.16: Validation plot for 24 hour test with default configuration, $n = 2$, $t = 250\,\mu s$

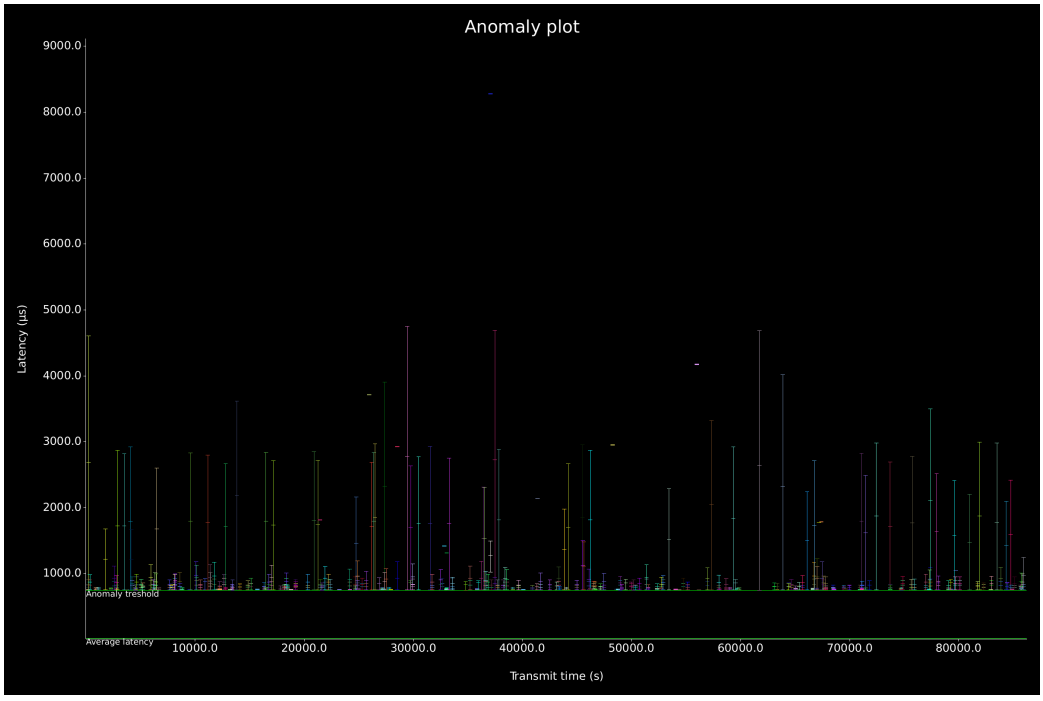

Figure 4.17: Validation plot for 24 hour test with default configuration, $n = 2$, $t = 750\,\mu s$

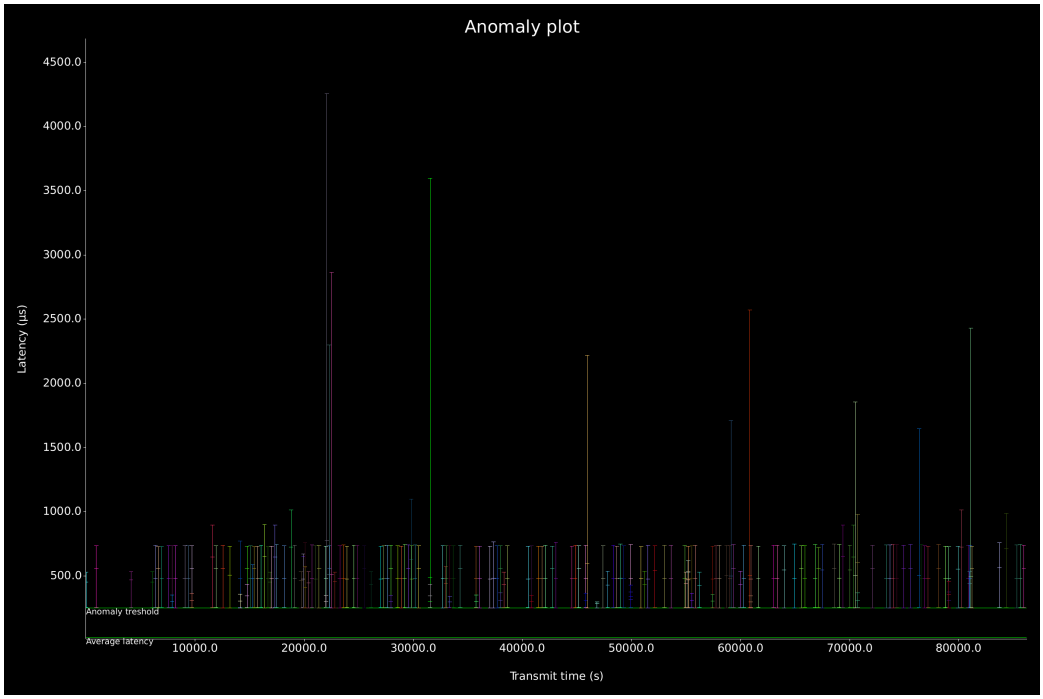

Figure 4.18: Validation plot for 24 hour test with modified configuration, $n = 2$, $t = 250\,\mu s$

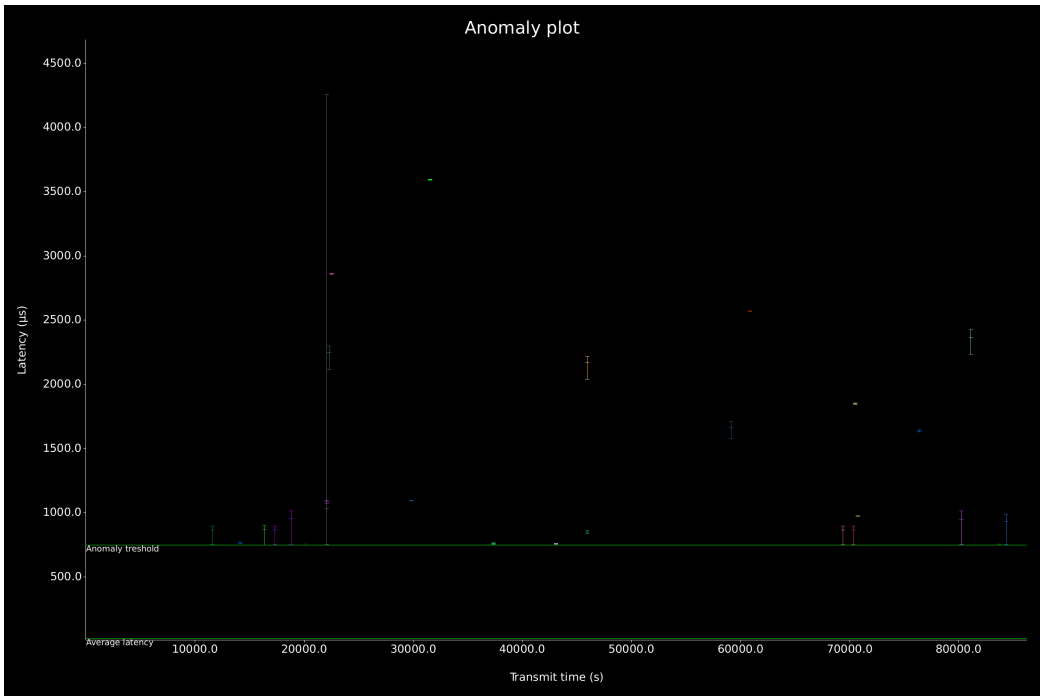

Figure 4.19: Validation plot for 24 hour test with modified configuration, $n = 2$, $t = 750\,\mu s$

less severe anomalies that happen more frequently. For example, there was a severe anomaly around 10 hours into the first test, where the maximum latency was over 8 ms. Anomalies like these would not have been visible in a test based on aggregated data, as the average latency would not have been visibly impacted by a single spike like this.

These results show a clear improvement in the amount of anomalies over a 24-hour period with the real-time kernel and halved queue sizes. There were fewer anomalies, and the maximum latency of anomalies was lower. Especially with the $t = 250\mu s$ threshold, the amount of anomalies was reduced by as much as 99.99%.

4.5 Discussion

Now that we have presented the results of our tests, we will discuss the results and their implications. We will also discuss the limitations of our tests, and how they could be improved.

Starting with the packet loss results for the 1700 kpps tests (Figure 4.5), we see that the configurations with threaded NAPI had much higher packet loss than the configurations without threaded NAPI. This is not what we expected, as we thought that threaded NAPI would help offload packet handling to dedicated CPU cores, allowing the kernel to prioritize the handling of packets. One potential reason for this, is that we tested by isolating the threaded NAPI to four CPU cores, while the non-threaded NAPI was allowed to use all CPU cores. So letting the system use all cores some of the time seems to work better than giving it exclusive access to a few of the cores.

Continuing with the latency results, we see that the runs with low traffic had much higher average latency compared to the runs with higher traffic rates. It is not clear why this happens, but one potential reason could be that the traffic is too low to trigger the usage of the NAPI, causing all packets to be handled by software interrupts instead. This would result in higher latencies, as the CPU would have to do more work for each packet. This applied to all configurations, where the results were very similar for all of them. All configurations also seem to be able to handle 19 kpps without significant impact on average latency, where all the configurations had a similar, low average value of between 20 and 25 µs.

At 190 kpps, both the average latency and the number of anomalies start to increase a lot, especially for some of the configurations. Average latency and anomaly counts are closely related as they both increase, since the packets that are part of anomalies will pull up the average value.

Most of the tests are relatively stable between the 15 runs, with narrow confidence

intervals. However, a major limitation of our testing was that we did all the 15 tests in a row, without rebooting the machine between tests. This means that the results could be affected by the state of the system, and that the results could have been different if we had randomized the ordering a bit, or rebooted more often. We did reboot between all configuration changes, so all tests were started from a fresh startup.

Our results do in general display a disappointing lack of patterns. We expected to see some configurations perform better than others, or at least some parameters affecting things consistently. Unfortunately, the results are very inconsistent, and it is hard to draw any conclusions from them. It could potentially have helped to do a reboot of the system for every five tests or similar. Thankfully, it is possible to look for improvements on a case-by-case basis, for example for our 190 kpps traffic rate.

The 24-hour tests are easier to compare, since we only have two test runs to compare. We see a clear improvement on the anomaly count and maximum latency when using the real-time kernel and halved queue sizes. With the parameters $n = 2$ and $t = 750\,\mu s$, the real-time kernel with halved queue sizes had only 27 anomalies over a 24-hour period, which is very low, and could be usable for a real-time system.

In summary, it is not possible to draw any general conclusions from our results, but we can see that by default, the Linux kernel performs well. The default configuration had no outstandingly bad results, and the average latency was low. However, there were still variations within the sub-millisecond range, and the extreme cases would still reach up to 8 ms. The real-time kernel seems to offer some improvement in special cases, but it has to be tested for each individual case.

4.6 Sources of error and limitations

There are several potential sources of errors in our tests. We will discuss some of them here, and how they could have affected our results.

As mentioned in the previous section, we did not reboot the system between all tests. This could have affected the results, as the state of the system could have been different between some of the tests. Ideally, this should not have been a problem, as we did reboot between all configuration changes, so all tests were started from the same state. However, with a complex real-world system, it is hard to guarantee that the state is exactly the same between all tests.

A second potential source of error is that we did not properly account for the latency induced by TRex. As discussed in subsection 3.3.2, we did spend time on minimizing the amount of latency spikes caused by TRex, but we have not subtracted the latency caused by TRex from our results. This means that the results for latency

do not show the exact latency of the system, but rather the latency of the system plus the latency of TRex. This is not a major issue, as the latency of TRex is relatively low and can be assumed to be constant across all tests. However, it is something to keep in mind when looking at the results.

A limitation to our tests, is that we only tested with a single traffic rate, due to our difficulty with adding support for multiple packet sizes. This means that we do not know how the system would handle fewer, but larger packets. It is possible that the system would perform better with fewer, larger packets, as the CPU would have to process fewer headers.

Chapter 5
Conclusion

In this thesis we have presented a benchmarking testbed for measuring the performance of network devices in the context of real-time applications, as well as used the testbed on the Linux kernel in different configurations. We started by looking into the current state of the art in the field of network device benchmarking, and motivated the need for an open source solution. We then presented the design of the testbed, and how we used it. Finally, we described various experiments we performed on the Linux kernel, and presented the results.

Our testbed seems to work well, and can be reused in the future for other projects with similar needs to ours. We have not tested the testbed on other hardware setups, but all the tools we used should be compatible with many other configurations. The testbed provides an answer to our first research question, which was about how to design an open-source benchmarking testbed for real-time applications.

Whether we have answered our second research question, is up for debate. The question was about what the performance bottlenecks of the Linux kernel are, and how they can be improved. Our results did unfortunately not provide any clear answers to this question, so further research is needed to fully answer this complex question. We did however find some interesting results, which can be used by future researchers to further investigate the performance of the Linux kernel.

Continuing on this, we do not have a clear answer to our third research question either. The question asked what would be the optimal configuration of the Linux kernel for real-time applications. We did find that for a specific traffic type, using the real-time Linux kernel and halving the queue size would provide a clear improvement over the default configuration. However, this result was not generalized to different traffic rates.

Overall, the thesis was mostly a success. We did produce and document our testbed, which can be used by future researchers to further investigate the performance

of the Linux kernel or other network devices. We also found some interesting results, which could be used as a starting point for further research. We might have been a bit too ambitious with some of the research questions, but laid the groundwork for future research in this area.

If we were going to change something about the thesis, we would probably have given MoonGen a try for the generation part of the testbed. MoonGen does not support timestamping on as many network cards as TRex do, but could potentially have made it easier to generate traffic with customizable packet sizes. For the testing, we would have been more careful to do more reboots between test repetitions, to avoid potential errors due to the kernel not being in a clean state.

Further research into this topic could be to investigate what is causing the performance differences, by looking into the source code of the Linux kernel, or by profiling the system while the tests are running. Another interesting research topic would be to investigate the performance of other network devices, such as a typical home router, and compare it to the performance of the Linux kernel.

References

[1] ArchWiki. *Realtime kernel patchset*. 2023. URL: https://wiki.archlinux.org/title/Realtime_kernel_patchset (last visited: May 9, 2023).

[2] Bradner and McQuaid. *Benchmarking Methodology for Network Interconnect Devices*. RFC 2544. Mar. 1999. URL: https://www.rfc-editor.org/info/rfc2544.

[3] Scott O. Bradner, Kevin Dubray, et al. *Applicability Statement for RFC 2544: Use on Production Networks Considered Harmful*. RFC 6815. Nov. 2012. URL: https://www.rfc-editor.org/info/rfc6815.

[4] Jae-Geun Cha and Sun Wook Kim. "Design and Evaluation of Container-based Networking for Low-latency Edge Services". In: *2021 International Conference on Information and Communication Technology Convergence (ICTC)*. 2021, pp. 1287–1289.

[5] Cisco Systems Traffic Generators. *TRex - Realistic Traffic Generator*. 2015. URL: https://trex-tgn.cisco.com/ (last visited: Feb. 19, 2023).

[6] clap-rs. *clap: Command Line Argument Parser for Rust*. 2023. URL: https://github.com/clap-rs/clap (last visited: May 1, 2023).

[7] Tencent Cloud. *F-Stack*. 2023. URL: http://www.f-stack.org/ (last visited: Apr. 13, 2023).

[8] The kernel development community. *Linux Base Driver for the Intel(R) Ethernet Controller 700 Series*. 2018. URL: https://www.kernel.org/doc/html/v5.12/networking/device_drivers/ethernet/intel/i40e.html?highlight=i40e (last visited: Mar. 29, 2023).

[9] Jonathan Corbet. *NAPI polling in kernel threads*. 2020. URL: https://lwn.net/Articles/833840/ (last visited: May 9, 2023).

[10] The Rust Project Developers. *Understanding Ownership*. 2023. URL: https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html (last visited: May 20, 2023).

[11] DPDK. *Data Plane Development Kit*. 2023. URL: https://github.com/DPDK/dpdk (last visited: Apr. 12, 2023).

[12] Eskil Dybvik. *Open source benchmarking of the Linux kernel network stack for real-time applications*. Project report in TTM4502. Department of Information Security, Communication Technology , NTNU – Norwegian University of Science, and Technology, Dec. 2022.

[13] Paul Emmerich. *MoonGen Packet Generator*. 2023. URL: https://github.com/emmericp/MoonGen (last visited: May 4, 2023).

[14] Paul Emmerich, Sebastian Gallenmüller, et al. "MoonGen: A Scriptable High-Speed Packet Generator". In: *Internet Measurement Conference 2015 (IMC'15)*. Tokyo, Japan, Oct. 2015.

[15] Paul Emmerich, Daniel Raumer, et al. "A study of network stack latency for game servers". In: *2014 13th Annual Workshop on Network and Systems Support for Games*. 2014, pp. 1–6.

[16] ESnet. *iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool*. 2023. URL: https://github.com/esnet/iperf (last visited: May 27, 2023).

[17] *ethtool(8) - Linux man page*. URL: https://linux.die.net/man/8/ethtool (last visited: Apr. 12, 2023).

[18] The Linux Foundation. *Intro to Real-Time Linux for Embedded Developers*. Mar. 21, 2013. URL: https://www.linuxfoundation.org/blog/blog/intro-to-real-time-linux-for-embedded-developers (last visited: May 28, 2023).

[19] The Linux Foundation. *What is Linux?* 2022. URL: https://www.linux.com/what-is-linux/ (last visited: Nov. 5, 2022).

[20] Sebastian Gallenmuller, Paul Emmerich, et al. "Comparison of frameworks for high-performance packet IO". In: *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS). Oakland, CA: IEEE, May 2015, pp. 29–38. URL: http://ieeexplore.ieee.org/document/7110118/ (last visited: Oct. 22, 2022).

[21] Open Traffic Generator. *Ixia-C Traffic Generator*. 2023. URL: https://github.com/open-traffic-generator/ixia-c (last visited: May 4, 2023).

[22] Bruno Giguère. *Are You Still Testing to RFC 2544? Really?* 2013. URL: https://www.exfo.com/en/resources/blog/still-testing-rfc-2544-really/ (last visited: Nov. 7, 2022).

[23] Ethan Grossman. *Deterministic Networking Use Cases*. RFC 8578. May 2019. URL: https://www.rfc-editor.org/info/rfc8578.

[24] Hanoch Haim. *Cisco TRex Manual*. 2023. URL: https://trex-tgn.cisco.com/trex/doc/trex_manual.html#_basic_usage (last visited: Feb. 18, 2023).

[25] Frank Alexander Kraemer. *Introduction to Design Science*. 2020. URL: https://falkr.github.io/designscience/preparation.html (last visited: Nov. 12, 2022).

[26] Stanislav Lange, Anh Nguyen-Ngoc, et al. "Performance benchmarking of a software-based LTE SGW". In: *2015 11th International Conference on Network and Service Management (CNSM)*. 2015, pp. 378–383.

[27] luigirizzo. *Netmap: a framework for fast packet I/O*. 2022. URL: https://github.com/luigirizzo/netmap (last visited: Oct. 23, 2022).

[28] ntop. *PF_RING ZC (Zero Copy)*. 2022. URL: https://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/ (last visited: Oct. 23, 2022).

[29] Srivats P. *Ostinato Traffic Generator for Network Engineers*. 2023. URL: https://ostinato.org/ (last visited: May 4, 2023).

[30] packagecloud. *Monitoring and Tuning the Linux Networking Stack: Receiving Data | Packagecloud Blog*. 2016. URL: https://blog.packagecloud.io/monitoring-tuning-linux-networking-stack-receiving-data/ (last visited: Oct. 4, 2022).

[31] Michal Pal. *Using Avalanche vs. TRex to Simulate CDN and Attack Traffic*. 2016. URL: https://www.imperva.com/blog/trex-traffic-generator-software/ (last visited: May 27, 2023).

[32] plotters-rs. *Plotters - A Rust drawing library focusing on data plotting for both WASM and native applications*. 2023. URL: https://github.com/plotters-rs/plotters (last visited: May 1, 2023).

[33] DPDK Project. *DPDK homepage*. DPDK. 2022. URL: https://www.dpdk.org/ (last visited: Oct. 10, 2022).

[34] Kristof Provost. "The Humble Bridge". In: *FreeBSD Journal* (March/April 2020), p. 6.

[35] Bolla Raffaele and Roberto Bruschi. "Linux Software Router: Data Plane Optimization and Performance Evaluation". In: *Journal of Networks* 2 (June 2007).

[36] Scapy community. *Scapy*. 2023. URL: https://scapy.net/ (last visited: May 19, 2023).

[37] Dan Siemon. 2013. URL: https://www.coverfire.com/articles/queueing-in-the-linux-network-stack/ (last visited: May 28, 2023).

[38] Agilent Technologies. *Agilent Technologies Spins Off Its Electronic Measurement Business, Keysight Technologies*. 2014. URL: https://www.investor.agilent.com/news-and-events/news/news-details/2014/Agilent-Technologies-Spins-Off-Its-Electronic-Measurement-Business-Keysight-Technologies/default.aspx (last visited: May 27, 2023).

[39] Junghan Yoon, Jian Li, and Sangho Shin. "A Measurement Study on Evaluating Container Network Performance for Edge Computing". In: *2020 21st Asia-Pacific Network Operations and Management Symposium (APNOMS)*. 2020, pp. 345–348.

Appendix A
Testbed guide

This appendix contains the guide we wrote for how to recreate the testbed. It is also available on our GitHub repository.

A.1 Creating a testbed for testing network devices suitability for real-time applications

These documents describe how to re-create the testbed I used in my master's thesis. The testbed can be used to see how well a device can handle real-time applications, by measuring "anomalies" in traffic handling. An anomaly is defined as "A group of N consecutive packets that have a latency over a threshold T".

The guide has several parts, split into multiple sections. To have a full testbed setup, it is recommended to follow them in order. The parts are:

1. Introduction: (this section)
2. Hardware Setup: Hardware requirements, and how to set it up.
3. Packet Generator software setup: Software installation and configuration on the packet generator.
4. Device Under Test setup: Software installation and configuration on the device under test. Contains the general procedure, and a specific example for a Linux desktop.
5. Test running and analysis: How to run the tests, and how to analyze the results.
6. Packet Generator validation and tuning: How to validate the packet generator setup, and how to tune it for optimal performance.

To get started, continue to the Hardware Setup section.

A.2 Hardware setup

The testbed uses two computers, one to generate traffic (the Packet GENerator, PGEN), and one device to be tested (DUT). Both devices have some minimum requirements to be able to work under this test setup. For the PGEN, the requirements are:

- A network interface card with at least two interfaces, that are supported by Cisco TRex. See table 5 on this page of the TRex manual[1] for a list of supported network cards.
- An installation of a Linux distribution. A fresh installation of Arch Linux was used during testing, but other distributions should work too.
- In addition, it is recommended to have at least 32 GB of RAM, and a "powerful" desktop CPU. An Intel Xeon W-1270P running at 5.10 GHz was used during testing.

For the device under test, the requirements are:

- Two network interfaces
- The ability to route or forward packets between them

During testing, two point-to-point connections should be made between the PGEN and the DUT, so traffic can flow both ways through different interfaces.

Once you have the hardware setup ready, continue to the Packet Generator software setup section.

A.3 Packet Generator software setup

This section details how to set up the packet generator for performing tests.

A.3.1 Initial setup

These are the steps needed for creating the traffic generator for the first time.

1. Install a Linux distro on the PGEN. Arch Linux is used in this example.

[1] https://trex-tgn.cisco.com/trex/doc/trex_manual.html#_hardware_recommendations

2. Install `git` and GCC (if you receive errors later, install `gcc8` and try again)
3. Clone the trex-core fork: `git clone https://github.com/KHTangent/trex-core`
4. Build TRex:

   ```
   cd trex-core/linux_dpdk
   # Either
   ./b configure
   ./b build
   # Or, in case the above commands give errors
   CXX=g++-8 CC=gcc-8 ./b configure
   ./b build
   ```

5. Find the ID's of the network cards you want to use.

   - cd into the scripts directory of TRex: `cd ../scripts`
   - Run `sudo ./dpdk_setup_ports.py -s` to see a list of available ports.

     ```
     Network devices using kernel driver
     ===================================
     0000:00:1f.6 'Ethernet Connection (11) I219-LM' if=eno1 ...
     0000:01:00.0 'Ethernet Controller X710 for 10GBASE-T' if=enp1s0f0 ...
     0000:01:00.1 'Ethernet Controller X710 for 10GBASE-T' if=enp1s0f1 ...
     ```

     In our case, we want to use the Intel X710 interfaces, which here have ID's `01:00.0` and `01:00.1`

6. Create a TRex configuration file somewhere, with the following contents. Replace the port ID's with the ones you found in the previous step.

```
- port_limit: 2
  version: 2
  interfaces: ["01:00.0", "01:00.1"] # Replace if needed
  port_info:
  - ip         : 11.11.11.2
    default_gw : 11.11.11.1
  - ip         : 12.12.12.2
    default_gw : 12.12.12.1
```

TRex is now installed and ready to use.

A.3.2 Initializing TRex

In addition to the initial setup, the port configuration must be applied after every reboot by using the `dpdk_configure_ports.py` script:

```
cd trex-core/scripts
sudo ./dpdk_setup_ports.py --cfg path/to/config.yaml
```

TRex is now functional, and can be used as-is. Continue to the Device Under Test setup section to set up the device under test.

After making sure TRex works, it is recommended to spend some time on tuning the setup. This is described in the final section, Packet Generator validation and tuning.

A.4 Device Under Test software setup

No matter what the device under test is, it must follow a few requirements:

- Have two or more network interfaces
- Be able to route packets between them

We used a Linux desktop during our testing, but any device that meets the above requirements can be used. We will give a general procedure for configuration of the DUT, then show how we accomplished it on our Arch Linux desktop.

A.4.1 General procedure

1. Enable forwarding of packets on the device
2. Enable the two interfaces
3. Add static routes for the two interfaces used by TRex. In the previous section, we configured TRex to use `11.11.11.2` as its IP address on one interface, and expects the DUT to use `11.11.11.1` on the same interface. The same applies to the other interface, where the PGEN uses `12.12.12.2` and expects the DUT to use `12.12.12.1`. Therefore, assign a static IP address of `11.11.11.1/24` to the interface that will be used for receiving data, and `12.12.12.1/24` for the other interface.
4. TRex will send packets with a source IP address in the `48.0.0.0/8` subnet, with a destination address in `16.0.0.0/8`. Therefore, add static routes for these subnets, with the PGEN as the gateway.

5. Sometimes, static ARP routes must be added on the DUT to locate the PGEN. Find the MAC addresses of the interfaces on the PGEN, and add static ARP routes for them on the DUT.

A.4.2 Example: Linux desktop

If the DUT is a Linux device, it can be helpful to put all the required setup steps in a shell script, since many of them will have to be re-executed after every reboot. The script could look like this:

```
#!/bin/bash
# Enable forwarding of packets
sysctl -w net.ipv4.ip_forward=1

# Bring up our interfaces. Replace with actual interface names
ip link set enp1s0f0 up
ip link set enp1s0f1 up

# Add static IP addresses to the interfaces. Replace the
# interface names if needed
ip a add 11.11.11.1/24 dev enp1s0f0
ip a add 12.12.12.1/24 dev enp1s0f1

# Add static routes for the traffic from TRex. These commands
# should work as-is, if you use the config given above
ip route add 48.0.0.0/8 via 12.12.12.2
ip route add 16.0.0.0/8 via 11.11.11.2

# Add static ARP entries. Replace the MAC addresses with the
# ones of the PGEN interfaces
arp -s 11.11.11.2 68:05:ca:df:09:26
arp -s 12.12.12.2 68:05:ca:df:09:27
```

Run this script as root after every reboot, and the DUT should be ready to receive traffic from TRex.

Now that the DUT is ready, it is finally time to run some tests. Continue to the Running the tests section.

A.5 Test running and analysis

This section describes how to run the tests, and how to analyze the results. We will start by preparing the `data-postprocessor`, which will be used to analyze the results obtained from TRex.

A.5.1 Data-postprocessor setup

This can be performed on any machine, for example on the PGEN, or a separate device. Using the PGEN can be convenient, as the data files can get quite large.

1. Install Rust. The simplest way is usually to use `rustup`[2]

   ```
   curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
   ```

2. Clone this repo, and build the data-processor

   ```
   git clone https://github.com/KHTangent/thesis-subprojects
   cd thesis-subprojects/data-postprocessor
   cargo build --release
   ```

3. If you want to install the data-postprocessor system wide, run the following command:

   ```
   cargo install --bins --path .
   ```

If you chose to not install the executable system wide, it will be located in `thesis-subprojects/data-postprocessor/target/release/data-postprocessor`. It can be freely moved to a more convenient location if you want, it does not have any external dependencies.

A.5.2 Running a test

Now that you have configured your PGEN and your DUT, you can run a test. A general test has three stages:

1. Configure the DUT to forward packets between the two interfaces, and apply any other configurations you want to try on the DUT.
2. Use TRex to generate latency measurement traffic, which is stored as a data blob.

[2] https://rustup.rs/

3. Analyzing this blob using the data-postprocessor.

To run a test, run the commands below on the PGEN. Remember to initialize your ports if you have not done so already.

```
cd trex-core/scripts
sudo ./_t-rex-64 --cfg path/to/config.yaml --lo \
   -l 190000 -f cap2/dns.yaml -d 60
```

Explanation of parameters:

- `--lo` Send only latency traffic. Latency traffic is the only traffic we can obtain full latency stats for, so only send this
- `-l 190000` Send 190 thousand latency packets every second, giving about 100 Mbps of traffic
- `-f cap2/dns.yaml` TRex requires an input file to run the mode we use, but since we use `--lo`, the contents doesn't affect anything. `cap2/dns.yaml` is a simple minimal file.
- `-d 60` Run test for 60 seconds

After the test has finished, a data blob will be placed in your `trex-core/scripts` directory, titled `timestamps-[date]-p0`. This file contains raw values for transmit and arrival times of all latency packets generated by TRex. This file is accepted by the data-postprocessor.

A.5.3 Viewing results

The data blobs can be analyzed using the data-postprocessor. The data-postprocessor has a help page that can be viewed by running `data-postprocessor --help`.

Examples of commands that can be run:

```
# Print a summary of anomalies in the data blob, and save a
# plot to plot.png. Consider 2 consecutive packets with a latency
# of 500 µs an anomaly
# Cut away the first and last second of the data
data-postprocessor timestamps-[date]-p0 validate -n 2 -t 500 \
```

```
   --summary-only -c 1 -o plot.png

# Print all of the anomalies in the data blob, and save a plot
# to plot.png. Consider 2 consecutive packets with a latency of
# three times the average latency an anomaly
# Cut away the first and last five seconds of the data
data-postprocessor timestamps-[date]-p0 validate -n 2 -d 3 \
   --summary-only -c 5 -o plot.png

# Plot latencies of all packets in the data blob. Include all data
data-postprocessor timestamps-[date]-p0 plot -p latency -o plot.png
```

If you are able to run tests, it is recommended to spend some time on tuning and validating your setup. This is described in the last section, Tuning the PGEN.

A.6 Tuning the packet generator

This section gives some tips on how to tune the packet generator (PGEN) to get more accurate results.

A.6.1 Run TRex in loopback mode to reduce outside interference

While tuning the PGEN, it is recommended to run TRex in loopback mode. This means that the PGEN will send packets to itself, instead of sending them to the DUT. This has the advantage that the PGEN is not affected by the DUT's performance, and that the PGEN can be tuned without needing to have the DUT connected. Running in loopback mode requires a different configuration file than the one used for normal testing.

First, connect the two physical network interfaces on the PGEN to each other.

Create a copy of your TRex config file titled `loopback.yaml`, and set your IP addresses like this:

```
- port_limit: 2
  version: 2
  interfaces: ["01:00.0", "01:00.1"] # Replace if needed
  port_info:
  - ip         : 11.11.11.2
    default_gw : 12.12.12.2
  - ip         : 12.12.12.2
    default_gw : 11.11.11.2
```

Run the `dpdk_setup_ports.py` script as described earlier, but use this new `loopback.yaml` config file as input parameter. If an error occurs because the ports are already bound, reboot the PGEN, and try again.

```
cd trex-core/scripts
sudo ./dpdk_setup_ports.py --cfg path/to/loopback.yaml
```

A.6.2 Suggested optimizations

Even while running in loopback, it is possible that latency spikes appear in the test results. This can happen for various reasons, for example because of other processes running on the PGEN, or how TRex is implemented. To make results as accurate as possible, it is recommended to spend some time experimenting in loopback mode until you get a good baseline with minimal spikes. This section will give some suggestions.

Minimize the number of processes running on the PGEN

A good starting point is to disable all services that are not needed for the PGEN to function.

The commands in this section assume that you are using `systemd` to manage your services. If you are not, you will need to adapt them to your system.

First, use a tool like `htop` to get an overview of what's running on the system. Many services can be disabled by simply running `systemctl disable <service>`.

In addition, it is recommended to disable X11, and any other graphical services, on the PGEN. TRex is a command-line application, so a desktop environment is not needed for it to function. To disable X11, run the following command:

```
sudo systemctl set-default multi-user.target
```

Reboot, and you should be taken to a terminal instead of a graphical login screen.

To undo this change at a later point, use the following command:

```
sudo systemctl set-default graphical.target
```

Run TRex on an isolated CPU core.

1. Add the following to your GRUB configuration to isolate four CPU cores from the kernel:

```
isolcpus=0,1,2,3
```

2. Reboot
3. Make sure your CPU cores are isolated by checking the contents of `/sys/devices/system/cpu/isolated`
4. Prefix all TRex commands with `taskset -c 0-3`. For example, to run the test mentioned earlier:

```
taskset -c 0-3 sudo ./_t-rex-64 --cfg path/to/config.yaml --lo \
-l 190000 -f cap2/dns.yaml -d 60
```

Switch to the preempt-rt kernel

The preempt-rt kernel is a real-time kernel, which is designed to minimize jitter and spikes by changing how the kernel handles scheduling, among other things. In our experience, it has been very effective at reducing spikes from the PGEN. Switching to the preempt-rt kernel is a bit distribution-dependent, so the following steps are only directly applicable to Arch Linux.

1. Install the `linux-rt` package: `sudo pacman -S linux-rt`
2. Regenerate your GRUB configuration: `sudo grub-mkconfig -o /boot/grub/grub.cfg`
3. Reboot

Once the system has booted, you can verify that you are running the preempt-rt kernel by running `cat /proc/version`. It should contain `rt` in the output.