CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Estimate CPU Utilization for Data Processing with DPDK

Degree Project Report in Computer Science and Engineering

Anton Rydén and Erik Näslund

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Degree Project Report 2022

Estimate CPU Utilization for Data Processing with DPDK

Anton Rydén and Erik Näslund

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Estimate CPU Utilization for Data Processing with DPDK
Anton Rydén and Erik Näslund

Supervisor: Daniel Romell and Robert Linder Blomberg, Sandvine
Thesis advisor: Panagiotis Strikos, Department of Computer Science and Engineering
Examiner: Lars Svensson, Department of Computer Science and Engineering

Degree Project Report 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2022

Estimate CPU Utilization for Data Processing with DPDK
Anton Rydén and Erik Näslund,
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

In a world where networking and network systems are becoming more and more important, it is crucial that these systems are properly developed and monitored. The purpose of the project and aim is to create an algorithm that estimates the CPU workload during packet processing that is linear with respect to network traffic using the Data Plane Development kit (DPDK). With such a tool, it becomes possible to predict when a system might start dropping packets. An important part of the project is the design flow, which mainly consists of a single machine with NUMA design along with tools to gather and visualize the results. A challenge of this project is a lack of prior attempts, making it difficult to find a reference point. While a linear solution was found, it is not without its flaws. However, with further testing, a practical implementation can be achieved. One conclusion to the project is that it can serve as a baseline for future attempts and research.

Keywords: DPDK, data processing, load estimation, TRex, packet drops.

Acknowledgements

We would like to express our deepest gratitude to Daniel Romell and Robert Linder Blomberg for supervising the project and providing invaluable knowledge as well as guidance.

Thanks should also go to Sandvine for providing the interesting thesis and hardware.

We would like to extend our sincere thanks to Panagiotis Strikos for supervising the thesis writing.

Anton Rydén and Erik Näslund, Gothenburg, June 2022

Contents

List of Figures

List of Tables

1

Introduction

This chapter serves as an introduction to the degree project. It mainly introduces the problem, workflow, goal and challenges and the outline of the report.

When handling large volumes of network traffic in an application, it is possible to use the Data Plane Development Kit (DPDK) instead of using the much slower IP (internet protocol) stack that comes with the operating system. One reason is that DPDK contains data plane libraries that accelerate packet processing workloads for networking applications [1]. It can also increase packet processing performance by up to 10 times over a built-in IP stack [2]. Finally, DPDK primarily uses a Poll Mode Driver (PMD) which constantly polls a network interface waiting for new packets [3]. This means that the Central Processing Unit (CPU) usage for the process that polls the network card is continuously at 100% load. However, this number does not actually say anything about how much work was done, or how much margin there was before overloading the processor which causes the network application to lose packets. DPDK has a lot of different types of users, one example being a telecom operator. For a user, it is often very important to be able to supervise the "actual" load on the system in order to plan the capacity and avoid overloading the network. An easy estimation of the actual load would be, for example, measuring the amount of bursts containing packets compared to empty bursts.

This project was carried out together with the company Sandvine, which works with applications that handle large volumes of network traffic. They also provide application and network intelligence that helps its customers deliver optimized experiences to consumers and enterprises. Customers use Sandvines cloud-based solutions to analyse, optimize, and monetize application experiences using contextual machine learning-based insights and real-time actions. The office that provided supervision to the project is located in Varberg, Sweden. However, the project was not done on-site, but was instead worked on from a remote location. Supervision was primarily performed by Daniel Romell and Robert Linder Blomberg, Sandvine.

1.1 Defining the problem

Measuring the amount of bursts containing packets compared to empty bursts provides a rough estimate of the actual load, while often being far away from linear

with respect to network traffic. Examples of errors are, among others, batch-effects, which affect the load positively during high network traffic (receiving several packets in one batch during the same poll). Another example is non-linear effects on memory latency that affects the estimation negatively during high network traffic (high memory usage). A more accurate estimate for the load would need to take these, and other factors, into consideration.

1.2 Workflow

The project was split into weeks. Each week had a set number of tasks that needed to be executed. The weeks also consisted of one to two meetings per week, one to review the current week, and plan the upcoming week, and one to review documentation. The latter was optional, since each week had a varying amount of documentation. These meetings will have at least one supervisor attending to help, review and improve the process.

1.3 Goals, Challenges and Aim

The purpose of this project is to create an algorithm that estimates the CPU utilization that is linear with respect to the amount of network traffic. There are many benefits of a well-performing load estimation algorithm. One of the benefits is, that it would be possible to easily see how much of the resources are being utilized. This is important since the knowledge of how much overhead remains gives insight into when the system needs to be upgraded. Another benefit is the possibility to predict packet drops solely based on load estimation. The only delimitation of this project is that the development of the CPU load approximation algorithm is only for DPDK. Furthermore, the aim of this project is to:

- Investigate and account for factors that influence the precision of a simple load estimate.
- Develop an algorithm or heuristic method for load estimate that is linear with respect to network traffic.

One of the challenges of this project is the shortage of prior attempts or similar projects, which means that a potential solution or guidance will be difficult to find. Another challenge is determining the optimal parameters using DPDK such as, but not limited to, size of memory buffer, number of memory buffers, queue length and number of queues.

1.4 Outline

- Chapter 2 introduces the necessary theoretical material and the tools used.
- Chapter 3 clarifies the design flow, hardware and software setup.

- Chapter 4 encapsulates and evaluates the series of implementations resulting from the design flow.
- Chapter 5 brings a conclusion to the report with some discussions about the results.

2

Background

This chapter explains the necessary theoretical material and tools that were used in the project. It focuses mainly on the networking aspects, TRex and DPDK, along with some information about technologies necessary for the physical and virtual setup.

2.1 Networking

An essential part of networking are packets. They are units of data that are transmitted over a network, mainly consisting of user data and control information. Using packet switches, it is possible to send and receive packets across networks [5]. Usually, this can be done by sending bursts consisting of multiple packets. Moreover, defining packets as either Rx packets when they are received or Tx packets when they are transmitted makes it possible to count the amount that failed to reach its destination, commonly known as packet drops [6]. Each packet dropped is a lost piece of information. These packet drops can be calculated by taking the amount of Tx packets minus the amount of Rx packets. However, in order to reduce drops and ensure each packet is transmitted and received, it is possible to use a queue that forces each packet to wait for its moment to be transmitted, guaranteeing that packets are not transmitted simultaneously [5]. When the queue becomes full, it is possible that packets start dropping. Subsequently, the queue length can vary and be changed. Both long and short length have different advantages and disadvantages. The queue size has a significant effect on the system, since a large queue means that the probability of dropping due to the queue being full decreases at the cost of system performance, depending on the hardware. On the other hand, a smaller queue increases the chance of dropping, but improves system performance [7].

In order to enable computers and machines to partake in a network, it is necessary to have some sort of Network Interface Controller (NIC) installed. A NIC is a hardware unit with the sole purpose of enabling the machine to communicate to other devices through either Ethernet or Wi-Fi [8]. This communication makes it possible to transfer and receive data. When transferring data, it is moved to the NIC from the memory and vice versa when receiving [8]. However, depending on the NIC and system, this transfer of data can be done differently.

2.2 Computer Memory

Another factor that affects this project is computer memory. One of the relevant concepts are memory buffers. They are a temporarily reserved area of the memory with the main purpose of storing data when transferring it between two locations [8]. These locations can be either devices or applications. Also affecting the memory is memory allocation, a process where programs and services are assigned a set amount of virtual or physical memory space [9].

In order to achieve optimal performance with multiple processors in one system, it can be useful to utilize Non-Uniform Memory Access (NUMA). NUMA on Linux is a way to divide hardware resources into software abstractions known as "nodes". These nodes may contain zero or more CPUs. NUMA is also a design where the time to access the memory depends on the location relative to the processor [10]. Meaning, NUMA can be applied in order to create multiple machines with independent hardware out of a single system and make use of several CPUs to increase performance.

Another way to increase the performance of a system is increasing the size of Memory pages, which by default are blocks of fixed-length virtual memory. However, this length can be increased with HugePages, a feature integrated into the Linux kernel. The system resources required to access page table entries can be reduced by increasing the size of memory pages. Thereby improving system performance for certain scenarios [11]. However, one disadvantage of HugePages is the increased risk of internal fragmentation. Since the memory is reserved for the page, storing small amounts of data in a large memory page will cause the remaining memory to be wasted. Usually, memory pages have a default size of 4 KB, while HugePages can achieve much larger depending on the system.

2.3 Data Plane Development Kit (DPDK)

To comprehend how DPDK works, it is important to understand what a data plane is. Essentially, a data plane is the part of a network that handles and transmits user packets [12]. This means that DPDK is a tool to develop and change how packets are processed. Furthermore, DPDK is a collection of libraries aimed at speeding up packet processing workloads on various CPU architectures [1]. Developed by Intel in 2010, it is written in the programming language C and is open-source, and as such, it can be downloaded and modified to fit any network application.

An essential part of DPDK is its Poll Mode Driver (PMD), which consists of APIs that are provided by the Operating System's (OS) drivers. Its purpose is to allow uninterrupted access to the Rx and Tx descriptors in order to receive, process, and send packets in the application. [3]. Hence, it is a vital component of this project, since it enables packet processing.

In order to test the PMD, one can use the DPDK application TestPMD which en-

ables the user to test their PMD by forwarding packets across networks [4]. TestPMD comes with a number of different forwarding modes, listed as follows:

- Input/output mode - default mode, both receives and transmits packets.
- Rx-only mode - only receives packets.
- Tx-only mode - only transmits packets.

While enabling users to try different scenarios for development, the modes also make it possible to modify and add actions carried out during packet processing. However, depending on the packet forwarding engine, the packets can be forwarded differently [13]. For example, one forwarding engine could forward packets without processing the packets, while another might scan each packet for various information. Thus, the forwarding engine can affect the result and performance of a system.

In order to set up several instances of DPDK on one machine, it is possible to use a Virtual Machine (VM). A VM is a machine built on code that uses the host hardware in order to simulate a physical machine. Typically, creating a VM can be done by assigning it restricted access to the physical machine hardware [14]. For example, it is possible to only allow the VM to use 30% of the physical machine memory. From a number of applications that make it possible to create a VM, the one relevant for this project is Quick Emulator (QEMU), an open source emulator that aims to allow users to run programs and applications built for one machine, on another [15].

This project mainly used DPDK for developing the algorithm that is used to estimate the load and make sure that it is linear with respect to network traffic. Subsequently, Python 3 was used to create various diagrams.

2.4 TRex

TRex is an open-source traffic generator based on DPDK created by Cisco and is used to generate artificial traffic in a network [16]. Two of the main parameters that affect how TRex generates traffic is the traffic mix and multiplier. A traffic mix is essentially a model that describes the behaviour of traffic, while the multiplier can amplify the volume of traffic [17].

Before releasing networking applications, it can be meaningful to test the hardware and software using a traffic generator to see how the application performs during high and low traffic. TRex is crucial for this project in order to generate traffic and estimate load based on packets processed.

3

Design

This chapter aims to clarify the design flow, hardware and software setup. The design flow section explains how the problem was approached in order to explore possible solutions. Hardware setup describes what hardware was used and what problems occurred when using that hardware. Software setup clarifies what software was used and in what configuration, and difficulties that occurred with this setup.

3.1 Design flow

Initially, the hardware along with an operating system and necessary drivers were set up. Next, we continued by getting familiar with DPDK along with how to compile and run a DPDK networking application. When this application was running, the next step was to get TRex compiled and running. However, in order to have both TRex and DPDK running on the same system, it was necessary to use a VM. This and more about the setup is explained in further details in Sections 3.2 and 3.3.

After both applications were running, it was made sure that they were able to send traffic between each other by starting a DPDK networking application that forwards traffic. Subsequently, the TRex application was initiated to send traffic to the DPDK application, which would return the traffic to TRex. This was performed by starting TRex with a specified time. When this timer ends, it outputs a statistics summary of the run. In this summary, the most important statistics are TX and RX packets. More importantly, unless TRex has the same amount of TX and RX packets, it implies that the applications are experiencing packet drops. Meaning, it was possible to verify a successful setup by checking the TRex statistics. When testing, low traffic was initially used to confirm that the volume of traffic was not causing any problems. After the setup was working with low amount of traffic, higher volume of traffic was tested.

When the setup was done and confirmed to be working, development of various tools to help achieve the final results began. Primarily, two tools were developed, one for visualizing the load and packet drops in a graph using Python and the other being a bash script for running a series of TRex runs with increasing traffic to ensure consistency and creating a complete run. A TRex run was one initiation of the traffic generator, allowing it to run through a specified time frame.

After the setup and tools were fully functional, it was a matter of testing different solutions and testing what parameters affect performance. Whenever a test run was completed, the graphing tool was used to compare previous runs to analyse what difference the changes made. Depending on the results, appropriate changes were made to make the results closer to reality. Reality being the point of time when the applications start to continuously drop packets and the effective load is at 100%.

3.2 Hardware setup

All the hardware was facilitated by Sandvine, the company provided us free hands to choose between their available hardware. In the end, the Dell PowerEdge R730 was selected due to its high performance. A high performing computer would allow testing a wider range of volume in traffic, providing an opportunity for stress testing of possible solutions to a greater extent. After choosing the base server hardware, there were two setups that were considered. The first one was to have two independent machines and send traffic between these. The second setup was to have one machine with two NICs and send traffic between these. The latter setup was chosen to minimize the hardware used, complexity, and was also recommended by Sandvine. The hardware used during the project can be seen in Table 3.1 below presenting the model, NIC, CPU and Random Access Memory (RAM).

Model	Dell PowerEdge R730
NIC	2 x Mellanox CX-5 dual port 100 Gbps Ethernet
CPU	2 x Intel Xeon E5-2697 v3 (14 cores 28 threads Max: 2.6 GHz Min 1.2 GHz)
RAM	2 x 128 GB (8 x 16 GB) DDR4 2133 MHz

Table 3.1: Hardware specification

As seen in Table 3.1 there are two sets of CPUs and RAM, since the motherboard is designed with NUMA. This has the advantage of being able to replicate two machines with independent hardware running on the same system. Two NICs were used in order to send traffic between the machines. The two NICs used have 2 ports each, which can be seen in Figure 3.1 below.

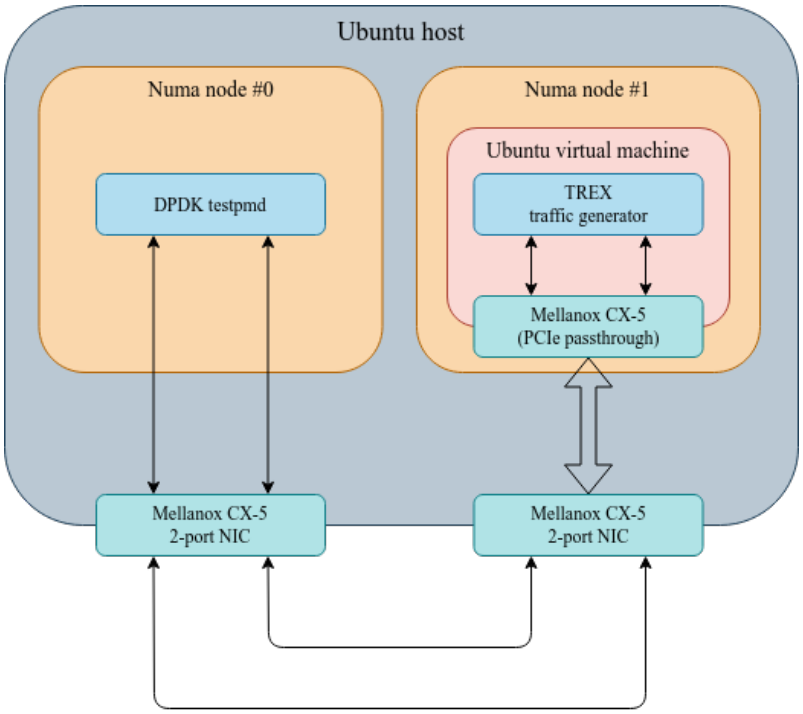

Figure 3.1: Hardware and software setup

There was one major discovery and challenge during the project with the NICs and motherboard. When running a high amount of traffic, there were highly inconsistent results. After some investigation, it was determined that the amount of transmitted packets from TRex and received packets from DPDK differed at larger volumes of traffic. None of the applications were actually responsible for the packet drops. The source of the problem was congestion on the NICs which could not absorb the amount of traffic coming from TRex. This type of drop is referred to as rx_discards_phy by Mellanox [18].

3.3 Software setup

As described in Section 3.2, the project was using one system with two NUMA nodes. These were utilized by running the DPDK applications on node 0 and the TRex traffic generator on node 1. The advantage of this is preventing the two applications from sharing resources. However, the setup proved to create some initial problems, since there were difficulties with running both DPDK and TRex on the same OS. These challenges appeared as a result of TRex being based on DPDK [16]. This caused complications since both applications are trying to launch one instance of DPDK each on the same OS. This was solved by running a VM with the help of QEMU on node 1 as seen in Figure 3.1. On node 0 the DPDK application was running and inside QEMU on node 1 the TRex was running, both the host OS and the VM running Ubuntu 20.04 Long Term Version (LTS).

After recommendation from supervisors, the selected application to run was TestPMD.

Initially, the default forwarding engine IO forward was used. This, in conjunction to the hardware, proved to be a bad setup for testing the algorithm. The reason was that IO forward does not access packets data, which makes this forwarding engine highly efficient [19]. So efficient in fact that testing the algorithm fully on one core was not possible and as such, it was not possible to send enough traffic to reach 100% utilization before the NICs were congested. This is explained in greater details in Section 3.2. There were two actions taken to resolve the issue. Firstly, underclocking NUMA node 0 to 1.2 GHz. This lead to worse performance of node 0 which resulted in being able to run full tests. However, it was still close to the limit of congestion, so in order to have more headroom, a second action was taken. This consisted of changing the forwarding engine to Checksum Only (CSUM), a forwarding engine that access the packet's checksum field and adjusting it [19]. Accessing and changing packet's data is much less efficient, which allows testing the algorithm fully.

Two tools were developed during the project, a graphing tool and a bash script. The graphing tool draws the results of runs, displaying both load and packets dropped. This tool proved to be very helpful when analysing results and comparing runs with each other. See Figure 3.2 below, demonstrating the output of the graphing tool.

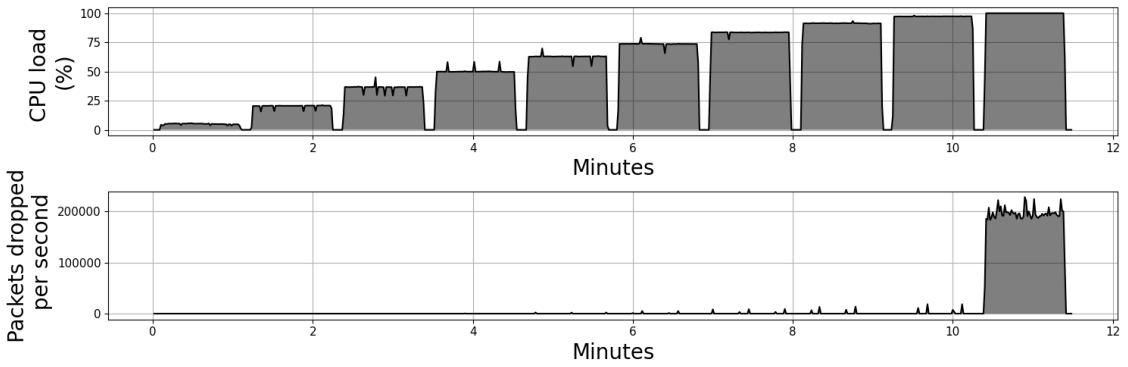

Figure 3.2: Example output for the graphing tool utilizing the bash script

The second tool developed was a bash script, which ensured there was consistency between runs. This was a necessary tool since it is not possible to gradually increase the amount of traffic during a run with TRex. Due to this, a bash script that consecutively started TRex processes with increased amount of traffic was constructed. When starting TRex, it is necessary to specify the amount of traffic in terms of multipliers, for example increasing the multiplier to 3 amplifies the amount of traffic specified in the traffic mix thrice. For the traffic mix used in the project, the base amount of traffic is roughly 1 Gigabits per second. This script took 4 arguments: lower bound, upper bound, time to run each process and increment of multiplier. For example, Figure 3.2 is displaying a run that make use of the tool. The specific arguments for this specific run were: 1 in lower bound, 51 in upper bound, 60 seconds per consecutive process and 5 multipliers increments.

4

Evaluation

This chapter encapsulates and evaluates the series of implementations resulting from the design flow. Each of which was based on a model created by Intel that calculates DPDKs Rx spin time [20]. See Equation 4.1 below, where *i0* represents the number of iterations that give zero packets, and *ti* the total number of iterations.

$$DPDK\ Rx\ Spin\ Time \propto \frac{i0}{ti} \tag{4.1}$$

Or in other words, the rate of packet receiving iterations that give zero packets out of all iterations. By changing this model slightly, it is possible to calculate the load of the CPU during packet processing. Since the iterations that receive packets are going to be the ones where the processor does any work of interest, we can change the model to focus on those iterations instead of those that give zero packets. See Equation 4.2 below, showing the updated model, where *i1* represents the number of iterations that give any packets.

$$CPU\ Load \propto \frac{i1}{ti} \tag{4.2}$$

While this model does not account for various factors that might affect each iteration, it is used as a reference point for each implementation. Each description of the implementations will also include a graph showing results and dropped packets. The results display the performance and drops of each solution. Thereby, they show how well it is possible to predict drops using the implementation. All results are generated by the same test, being one run through of the bash script described in Section 3.3 along with the parameters that provided the most reasonable results. See Table 4.1 for the parameters used in the test.

Packet Burst size	64
Rx queue size	4092
Tx queue size	4092
Number of Rx queues	1
Number of Tx queues	1
Number of CPU cores	1
Forwarding mode	CSUM
Memory buffer size	1646
Total number of memory buffers	200 0000

Table 4.1: Parameters used in the test that gave the results for each implementation

The parameters queue size, memory buffer size, and total number of memory buffers are a consequence of trial and error for finding the optimal DPDK performance. In this case, optimal performance is defined as when the networking application continuously drops packets at the largest possible traffic volume. It is reasonable to test the estimation algorithm on the best parameters for DPDK, since there is no real reason for having suboptimal performance in a real use case.

4.1 First implementation

Following the Equation 4.2 above, the project's first attempt was the direct implementation of that model. However, what exactly one iteration does, is not entirely elaborated. Therefore, this solution assumes that one iteration was one call of DPDKs forwarding engine function. Furthermore, since that function runs more than a hundred thousand times each second, the load was calculated one hundred thousand iterations apart to improve the time complexity and accuracy. See Figure 4.1 below, showing results of this implementation.

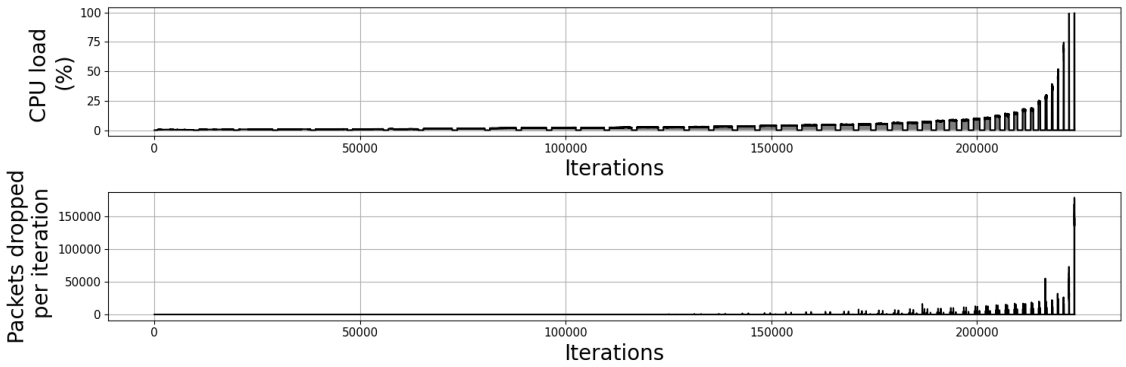

Figure 4.1: Load estimation and packet drops of first implementation

The implementation results in an exponential CPU load growth, which makes it difficult to predict drops. However, it starts dropping at around 100% efficient load.

4.2 Second implementation

In an attempt to improve the first implementation, a factor that could be affecting the result was removed. This factor was the fact that calculation of the load was made a set amount of iterations apart. Meaning that the efficient load was inconsistently calculated, since it depended on the time the CPU spent on each iteration. In order to prevent this, the second implementation made use of timestamps, to perform each calculation a set time apart. Hence, for the second implementation, the efficient load was calculated once every second. See Figure 4.2 below for the results and drops of the second implementation.

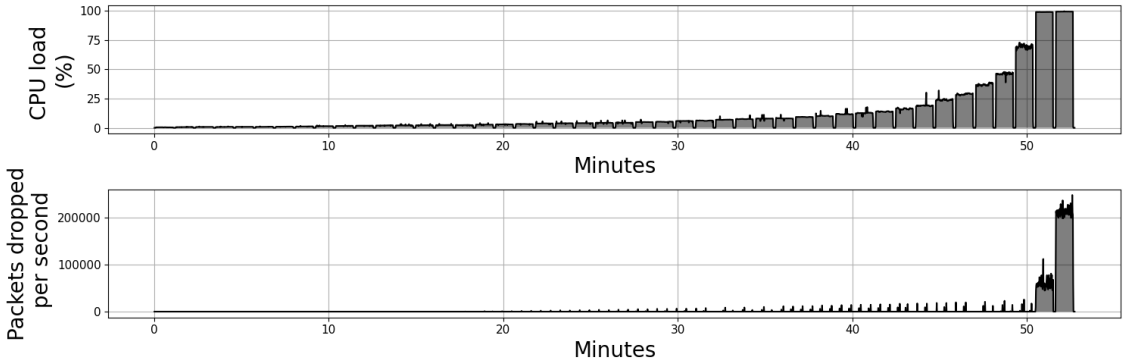

Figure 4.2: Load estimation and packet drops of second implementation

While possibly being more linear with respect to network traffic, the second implementation show some inconsistency and its CPU load still grows exponentially. However, it is possible to somewhat predict drops using the implementation.

4.3 Final implementation

While the use of timestamps improved the result slightly, it could be utilized to further develop the implementation. By replacing the inconsistent iteration factor completely and only focusing on the actual time spent on each calculation, the accuracy would improve significantly. This was achieved by removing the counting of each burst with any packets, and replacing it with counting the time spent on each burst. Thus, it would become possible to calculate the CPU load solely based on actual packet processing labour. However, this also means that the model was reworked. See Equation 4.3 showing the updated model, where *tb1* represents the total time spent on bursts with any packets and *tba* the total time spent on all bursts. Furthermore, see Figure 4.3 for the results of the final implementation.

$$CPU\ Load \propto \frac{tb1}{tba} \tag{4.3}$$

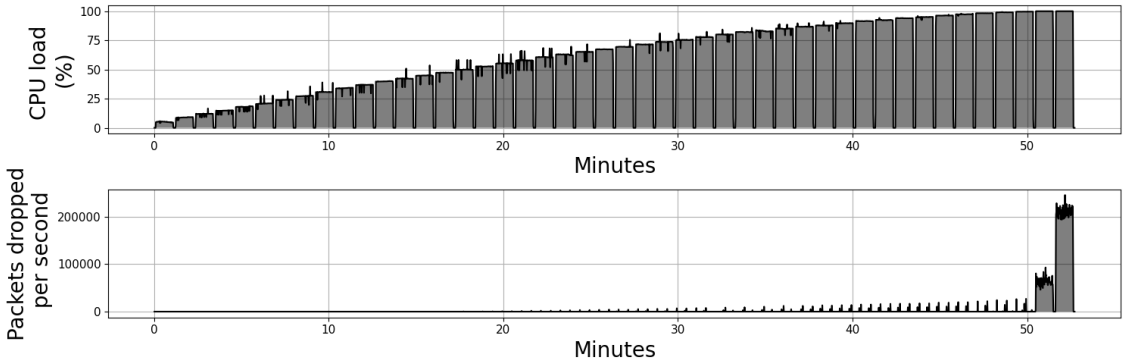

Figure 4.3: Load estimation and packet drops of final implementation

The results of the final implementation are evaluated in Section 4.4 by comparing it to a baseline and highlighting gathered observations.

4.4 Comparing final implementation with baseline

Since there are few prior attempts trying to solve this problem, it becomes difficult to find a reference point. Hence, the first implementation will be used as a baseline. Comparing the results from the first to the final implementation, there are a few obvious differences. See Figure 4.4, showing a vertically stacked comparison.

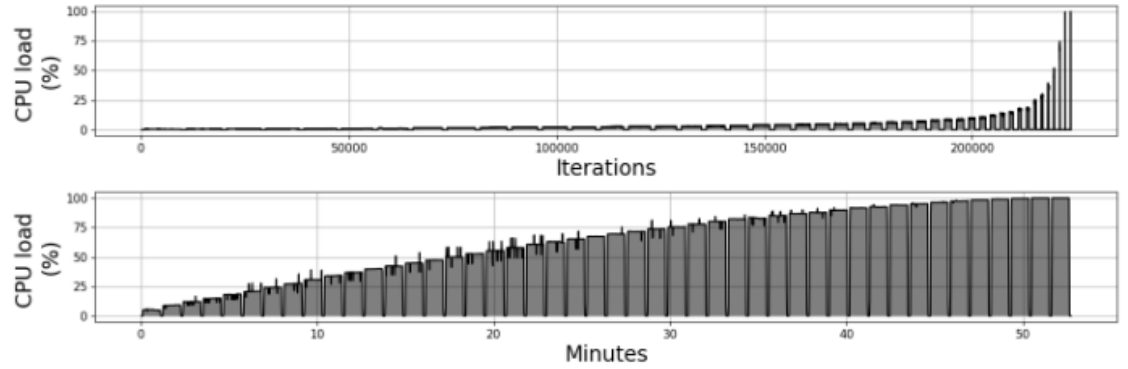

Figure 4.4: CPU Load, vertically stacked comparison, first implementation (above) vs final implementation (below)

Most prominent being the first implementation producing an exponential growth of load with respect to network traffic, while the final implementation is linear. Being linear, it becomes much easier to predict packet drops since each increase in multiplier provides a predictable outcome in load. Another observation is that the first implementation starts dropping at a lower CPU load. See Figure 4.5, showing a vertically stacked comparison of the performance in regard to packet drops.

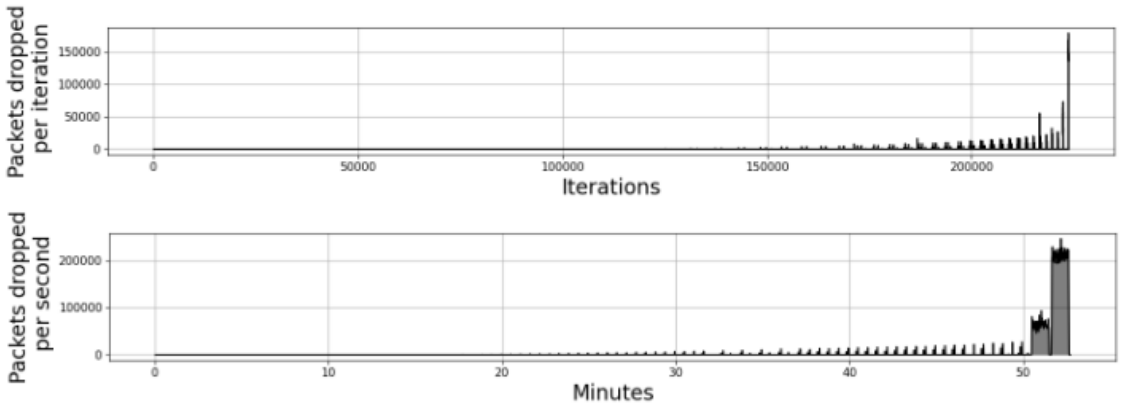

Figure 4.5: Vertically stacked Packet Drops comparison, first implementation (above) vs final implementation (below)

This indicates that the first implementation might require more processing power compared to the final implementation. However, we can also see the final implementation start to continuously drop packets at multiplier 45. Subsequently, it is difficult to further analyse and evaluate the drop comparison since the first solution is based on number of iterations, while the final implementation relies on time. Hence, below is an explanation that brings additional statistics of each solution, to clearly demonstrate the variance in performance.

Considering the aim was to create an estimation algorithm that is linear in respect to the networking traffic, a metric to use besides the graphs is necessary. Therefore, we created an equation that we call theoretical load, which calculates the expected load at a certain network volume in order for a solution to be linear. Thus, it is possible to build a table of statistics that shows how each implementation compares to a perfectly linear solution. This can be done by taking the traffic volume of the desired point, and dividing it with the measurement where the networking applications start to continuously drop packets. See Equation 4.4 below showing the created model, where *tvp* represents the traffic volume at the desired point in time and *tvd* the traffic volume when packets continuously drops.

$$Theoretical\ Load \propto \frac{tvp}{tvd} \tag{4.4}$$

An important fact about using this equation for our setup is that the used traffic mix increase the volume with increments of 1 Gbps for each multiplier. Thus, there is a chance that a multiplier lower than 45 causes consistent packet drops while having a load of 100%. However, we will use 45 as a reference point since it is the first multiplier that generated continuous drops in the tests. Furthermore, this means that the theoretical load is not an entirely perfect representation of a linear solution. Nonetheless, the theoretical load in this case, represents what load a linear solution would provide, given the fact that a traffic volume of 45 Gbps is the lowest volume that causes the implementation to reach 100% load and start dropping packets

consistently. See Table 4.2 for a comparison between the theoretical load and the estimated load of each implementation, and Table 4.3 showing packet drops for each implementation during the selected interval.

Metrics		Estimated load for each implementation		
Traffic volume	Theoretical load	First	Second	Final
1 Gbps	2.2%	0.4 %	0.4 %	5.0 %
9 Gbps	20%	1.4 %	1.4 %	30.2 %
18 Gbps	40%	3.0 %	2.9 %	54.8 %
27 Gbps	60%	6.0 %	5.8 %	75.3 %
36 Gbps	80%	13.0 %	12.7 %	91.5 %
45 Gbps	100%	98.3 %	97.1 %	100 %

Table 4.2: Theoretical load compared to the estimated load of the implementations

Metrics		Total amount of packet drops for each implementation		
Traffic volume	Theoretical load	First	Second	Final
1 Gbps	2.2%	0	0	0
9 Gbps	20%	0	0	0
18 Gbps	40%	2404	1818	1842
27 Gbps	60%	18842	14271	15761
36 Gbps	80%	50908	43925	45181
45 Gbps	100%	2894060	3300460	3531246

Table 4.3: Amount of packets dropped by each implementation

The results seen in Table 4.2 and 4.3 show that the final implementation have a greater performance in almost all aspects compared to the first and second solution,

having less packet drops before 100% load and being much more linear. However, it also demonstrates that improvements made from the first to the second implementation did not have a large impact with regard to linearity.

While the final implementation achieves better results than the first, it is not without flaws. Figures 4.4 and 4.5, and Table 4.2, show the inconsistencies of the solution. One being the visible spikes in each bar, as can be seen in Figure 4.3. Suggesting that there might be something that affects the load calculation. We believe that the spikes could be the result of batch effects, where a batch of packets is larger than the normal size. Since there are more calculations required during the processing of such a batch, the load becomes higher. However, it is difficult to prevent these effects in a real-world scenario since the traffic is not completely controlled. It is also not something that should be scheduled differently, since if the packets are not dealt with as fast as possible, they might be dropped, making it a difficult problem.

Another deviation is the sporadic drops, which are noticeable before the implementation starts dropping consistently. It is also apparent that the solution is not entirely linear in respect to networking traffic, looking at the results from Table 4.2 showing that the estimations are roughly 10% off. However, running the same test, we found that, compared to a non-modified instance of DPDK, the final implementation starts dropping about one multiplier earlier, negatively affecting the amount of dropped packets. Consequently, in order to create an authentic comparison, it is impossible to add all modifications that gather those results to visualize them in a graph. However, we believe that the cause of the early drops is most likely due to the time complexity of the algorithm. Reducing the processing power needed to run the algorithm should allow the processor to focus more on dealing with packets. This in itself would not only prevent how many packets drop, but also when the implementation starts dropping.

Parameters for DPDK make a substantial difference in performance and for the load estimation. In order to grasp what affects the performance of the networking application and the load estimation algorithm, we ran multiple tests with slight differences in parameters compared to 4.1. The results of decreasing queue size to 2048 shows roughly the same amount of drops per second during peak load, while beginning to drop small amounts earlier, which we believe is the cause of a worse ability of handling batch effects. See Figure 4.6 displaying the effects of decreasing queue size to 2048.

Moreover, when increasing the queue size to 8192, the solution generates earlier consistent drops and the amount of drops is significantly increased during peak load, as presented in Figure 4.7 below.

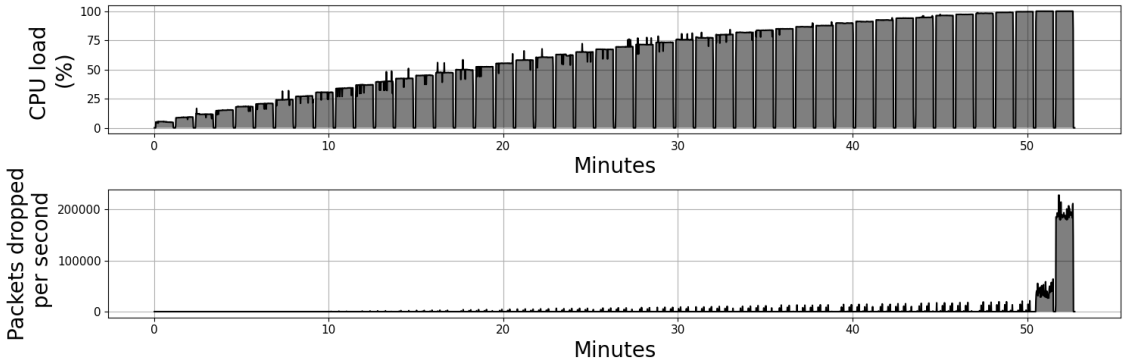

Figure 4.6: Results of final implementation with queue size 2048

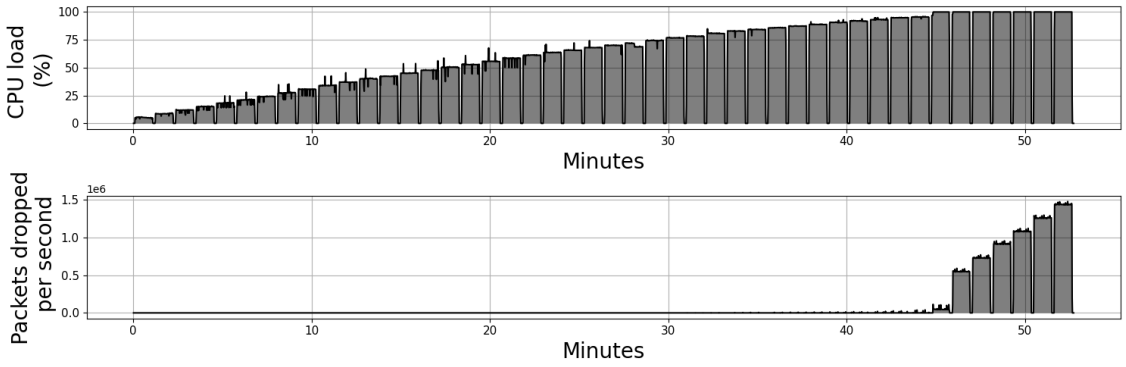

Figure 4.7: Results of final implementation with queue size 8192

This trend seems to continue when increasing the queue size further, as shown in Figure A.4 found in Appendix A, which we believe might be the result of an increased work load. However, when changing the total number of memory buffers, the results in Figure A.1 and A.2 shows no notable difference in performance. Furthermore, no remarkable changes in performance were observed when increasing the size of memory buffers, as visualized in Figure A.3. Figures A.1, A.2, and A.3 located in Appendix A.

5

Conclusion

This chapter brings a conclusion to the report. It also discusses the results, how they could potentially be improved, and what the next step in the project would be.

This project has been an attempt to create an algorithm that estimates the CPU load during packet processing that is linear with respect to network traffic based on DPDK. While successfully developing a substantially linear implementation, only the majority of factors that influence the precision in a simple load estimate were accounted for.

The results gathered were based on the design flow, which mainly consisted of a hardware and software setup. By using a single machine with NUMA design provided by Sandvine, it was possible to create a setup that had TRex and DPDK simulate a network system. However, before the setup was functioning, it was necessary to underclock the CPU and change the forwarding engine in order to be able to gather packet drops. Furthermore, creating tools to both test and present results was required to have an optimal setup. With a working setup that could gather results, where TRex was generating traffic and DPDK receiving and forwarding packets, numerous implementations that could estimate the CPU load during packet processing were created. This eventually led to a final solution that was based on calculation time.

An important note is that the results of the final implementation rely heavily on parameters. The parameters used to achieve the results were gathered through a trial and error process, by seeing how each individual setting affected the end result. Changing any of the parameters can cause a change in the end result. Altering the stability of the network application and therefore, load estimation, when packets start dropping and how many packets drop. On the other hand, results with unstable estimation were only found when the performance of DPDK was poor. In this case, we consider poor performance being when DPDK is only able to forward a substantially lower amount of traffic compared to optimal performance. Out of all the DPDK parameters that we tested during the trial error process, it appears that only the queue size drastically affect the result.

5.1 Future Work

Due to a lack of time, this project managed to create an incomplete solution and as such, there are more tests, investigations and research required to create a finalized implementation. Stress-testing the solution to see how it performs when the CPU is under additional pressure would be the next step. Using the algorithm in a real environment would most likely mean different applications and pieces of software running in the background. In addition, to further simulate how it would perform in such a setting, it would be beneficial to have two separate systems instead of having both TRex and DPDK on one machine. With the results from such tests, it would be possible to develop the algorithm to work for practical use. With more time, it would also be possible to further research how each previously mentioned parameter affects the end result and follow a more theoretical way of optimizing the test. This could potentially lead to better results and more insight in to what can improve the solution. However, since there were few prior attempts at solving this problem, we hope that this project will help further development of the issue and create a baseline.

Bibliography

[1] 'About DPDK' dpdk.org. https://www.dpdk.org/about/ (accessed Feb. 04, 2022).

[2] 'DPDK Boosts Packet Processing, Performance, and Throughput' intel.com. https://www.intel.com/content/www/us/en/communications/data-plane-development-kit.html (accessed Feb. 04, 2022).

[3] 'Poll Mode Driver — Data Plane Development Kit 21.11.0 documentation' doc.dpdk.org. https://doc.dpdk.org/guides/prog_guide/poll_mode_drv.html (accessed Feb. 04, 2022).

[4] 'Testing DPDK Performance and Features with TestPMD' intel.com. https://www.intel.com/content/www/us/en/developer/articles/technical/testing-dpdk-performance-and-features-with-testpmd.html (accessed May 09, 2022).

[5] J. F. Kurose and K. W. Ross '1.3.1 Packet Switching', In *Computer Networking: A Top Down Approach*, Seventh Edition, Boston: Pearson, 2017, pp. 23-26

[6] R. Ayanzadeh, E. Shahamatni, and S. Setayeshi, '1.4.2 Queuing Delay and Packet Loss', In *Computer Networking: A Top Down Approach*, Seventh Edition, Boston: Pearson, 2017, pp. 39-40

[7] J. F. Kurose and K. W. Ross 'Determining Optimum Queue Length in Computer Networks by Using Memetic Algorithms', *J. of Applied Sciences*, vol. 9, no. 15, pp. 2847-2851, Jul. 2009, doi: 10.3923/jas.2009.2847.2851.

[8] I. Englander, 'Computer Architecture and Hardware Operation', In *The Architecture of Computer Hardware, Systems Software and Networking: An Information Technology Approach*, Fourth Edition, Waltham, Massachusetts:

John Wiley & Sons Inc., 2004, pp. 178-367.

[9] I. Englander, 'File Management', In *The Architecture of Computer Hardware, Systems Software and Networking: An Information Technology Approach*, Fourth Edition, Waltham, Massachusetts: John Wiley & Sons Inc., 2004, pp. 548-591.

[10] 'What is NUMA? - The Linux Kernel documentation' kernel.org. https://www.kernel.org/doc/html/v5.9/vm/numa.html (accessed May 12, 2022).

[11] 'HugePages' docs.oracle.com. https://docs.oracle.com/database/121/UNXAR/appi_vlm.htm# (accessed May 12, 2022).

[12] J. F. Kurose and K. W. Ross '4.1.1 Forwarding and Rounting: The Data and Control Planes', In *Computer Networking: A Top Down Approach*, Seventh Edition, Boston: Pearson, 2017, pp. 306-310

[13] C. Wu and R. Buyya, 'Chapter 13 - Data Center Networks', in *Cloud Data Centers and Cost Modeling*, C. Wu and R. Buyya, Eds. Morgan Kaufmann, 2015, pp. 497-576. doi: 10.1016/B978-0-12-801413-4.00013-1.

[14] J. E. Smith and R. Nair, 'The architecture of virtual machines', *Computer*, vol. 38, no. 5, pp. 32-38, May 2005, doi: 10.1109/MC.2005.173.

[15] 'QEMU' wiki.qemu.org. https://wiki.qemu.org/Main_Page (accessed May 02, 2022).

[16] 'TRex' trex-tgn.cisco.com. https://trex-tgn.cisco.com/ (accessed May 02, 2022).

[17] 'TRex Book' trex-tgn.cisco.com. https://trex-tgn.cisco.com/trex/doc/trex_book.pdf (accessed May 10, 2022).

[18] 'Understanding mlx5 ethtool Counters' support.mellanox.com. https://support.mellanox.com/s/article/understanding-mlx5-ethtool-counters (accessed May 13, 2022).

[19] 'Testpmd Runtime Functions - Data Plane Development Kit 17.02.1 documentation' doc.dpdk.org. https://doc.dpdk.org/guides-17.02/testpmd_app_ug/testpmd_funcs.html

[20] 'Core Utilization in DPDK Apps' intel.com. https://www.intel.com/content/www/us/en/develop/documentation/vtune-cookbook/top/methodologies/core-utilization-in-dpdk-apps.html (accessed May 03, 2022).

A

Appendix

The figures below are the result of test runs with small modifications to the parameters found in Table 4.1. Each alteration is described in the figure label.

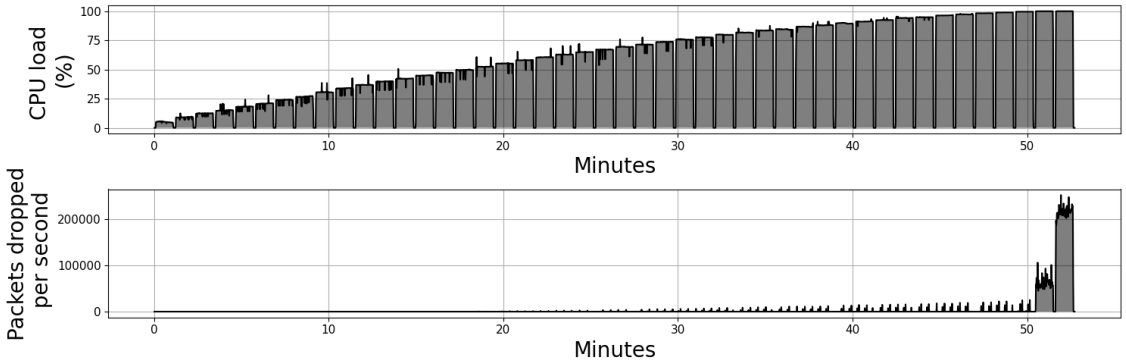

Figure A.1: Results of final implementation with total num mbufs 150000

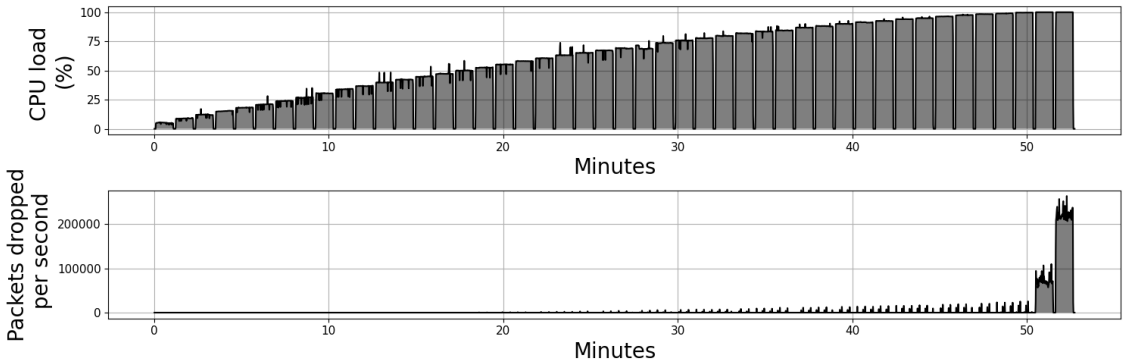

Figure A.2: Results of final implementation with total num mbufs 250000

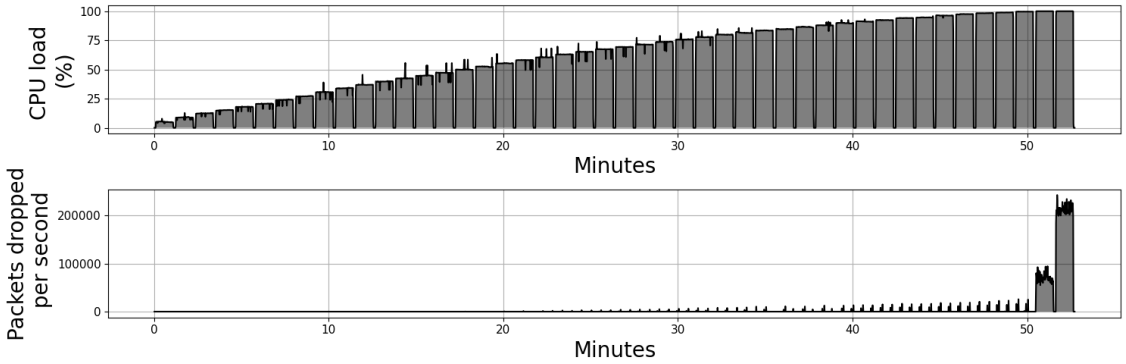

Figure A.3: Results of final implementation with mbuf size 2000

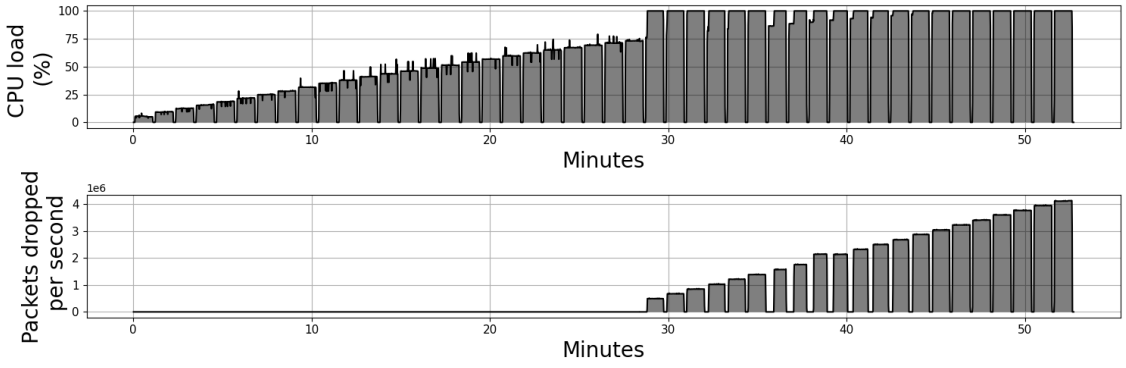

Figure A.4: Results of final implementation with queue size 16384