Development of a multi-threaded NAT

Hongyi Zhang <hongyiz@kth.se>
Lida Liu <lidal@kth.se>
Xuwei Gong <xuweig@kth.se>
Yuan Fan <yfan@kth.se>
Huanyu Wang <huanyu@kth.se>
Devika Suri Rohela <rohela@kth.se>

Final Report of Communication System Design

Communication Systems
School of Electrical Engineering and Computer Science (EECS)
KTH Royal Institute of Technology
Stockholm, Sweden

8 January 2019

Examiner: Professor Dejan Kostic
Academic Adviser: Tom Barbette

Abstract

Network Address Translation is used to remap one IP address space to another by modifying some fields in the IP header when packets are passing through. This technology enables a router to aggregate multiple private IP addresses into several public IP addresses or even one public IP address. Thanks to the IP address mapping technology used in NAT, the limited public IP addresses are available to more hosts. However, with the increasing number of mobile devices, a NAT router will encounter a performance bottleneck when it needs to deal with millions of packets per second. To solve this problem, a multi-threaded Carrier-grade NAT is proposed. It is practical as well as an essential technology for communication systems nowadays. Here, We divide our project into two parts: single-core NAT measurement and multi-threaded NAT development. Firstly, in order to easily measure different evaluation metrics of a single-core NAT, we set up an experimental environment by using a FastClick module. This gives us a clear view of the performance of NAT in use. We expect that latency is the critical parameter which leads to single-core NAT's bottleneck. On the contrary, after realizing multi-thread, the performance of NAT can be greatly improved. There are two methods which can be deployed for multi-threaded NAT implementation: Global Locked Table and Per-core Duplication. After comparison, per-core duplication shows more powerful and stable performance. In the final part, we will explore the source code of NAT and try to improve the performance of it fundamentally through a series of literature reviews, especially the data structure, HashTable and Heap Tree, used in FastClick.

Keywords:
NAT, FastClick, multi-thread

Sammanfattning

Network Address Translation används för att omkartlägga ett IP-adressutrymme till ett annat genom att ändra vissa områden i IP header när paketet går igenom. Tekniken möjliggör att en router kan samla in flera privata IP-adresser till flera offentliga IP-adresser, till och med en allmän IP-adress. Tack vare IP-adress kartläggningstekniken som används i NAT är de tillgängliga offentliga IP-adresserna begränsade. Men antalet mobilenheter har ökat, trots att problemet med otillräcklig offentlig adress är löst, kommer en NAT-router att stöta på en prestandaflaskhals när den behöver hantera miljontals paket per sekund. För att lösa detta problem föreslås en multi-threaded Carrier-grade NAT. Den är också praktisk som en väsentlig teknik för kommunikationssystem nuförtiden. Vi delar vårt projekt i två delar: enkelkärna NAT-mätning och multi-threaded NAT-utveckling. För att kunna mäta olika utvärderingsmetoder för en enkelkärna NAT på ett enkelt sätt, etablerar vi en experimentell miljö med hjälp av en FastClick-modul, vilket ger oss en tydlig bild av NATs prestanda. Vi förväntar oss att förseningen är den viktiga parametern som leder till enkelkärna NATs flaskhals. Däremot förbättras NAT-prestanda avsevärt när multi-threaded har genomfört. Det finns två metoder som kan användas för multi-threaded NAT-implementering: Global Locked Table och Per-core Duplication. Jämförelsen visar att Per-core Duplication har mer kraftfull och stabilare prestanda. I den sista delen kommer vi att utforska källkoden för NAT och försöker förbättra dess prestanda genom en rad granskning av vetenskaplig litteratur, i synnerhet datastrukturen, HashTable och Heap Tree som används i FastClick.

Nyckelord:
NAT, FastClick, multi-thread

Acknowledgements

We would like to thank our academic adviser Tom Barbette and examiner Professor Dejan Kostic. Tom's constant encouragement and guidance helped us stay on the right track. We also want to thank Dejan's fantastic IK2200 course providing us with a chance to this project, and Dejan's kind reminder whenever we were behind schedule.

Also, we would like to thank all fellows in our team for four months of hard work. We helped each other whenever we faced some complex problems. All of us have proposed some brilliant ideas and contributed to our project a lot.

Finally, we want to thank Cisco community's enthusiastic developers. They answered our questions about T-Rex efficiently and detailedly. We would have taken more time to complete our project without their assistance.

Stockholm, Jan. 2019
Team NAT

Contents

List of Figures

List of Tables

List of Acronyms and Abbreviations

This document requires readers to be familiar with terms and concepts described in RFC 1631[1] and RFC 2663 [2]. For clarity we summarize some of these terms and give a short description of them before presenting them in next sections.

IPv4	Internet Protocol version 4 (RFC 791 [3])
ICT	Information and Communication Technology
NAT	Network Address Translation
VM	Virtual Machine
CGN	Carrier Grade Network Address Translation
RFC	Request for Comments
DPDK	Data Plane Development Kit
ISP	Internet Service Provider
LSN	large-scale NAT
GUI	Graphical user interface
pps	packets per second
TX	Transmit
RX	Receive

Chapter 1

Introduction

Network Address Translation (NAT) is a technology which is widely deployed today. It aims to address the problem of the insufficient number of public IPv4 addresses. With the increasing expansion of today's networks, simply using traditional NAT can no longer fulfill Internet Service Provider's (ISP) requirements of deploying a large scale of network. This leads to the concept of Carrier Grade NAT (CGN). The main purpose of this project is to study different methods on how to build a CGN on a single host with multiple CPU cores. Starting with a single-core NAT realization, the project focuses on building a multi-threaded NAT in various ways to improve the performance of a single-core NAT.

Overview

Traditional NAT, which has been deployed on the Internet for decades, faces the depletion of public IPv4 addresses. Internet Service Providers (ISPs) have to seek new approaches to provide Internet service with limited public IPv4 addresses but increasing customers. One of the approaches turns out to be the CGNs, which follow the same idea of traditional NAT but are deployed by ISP in their internal network. In other words, ISPs can add another layer of IP address translation and aggregate multiple private IPv4 addresses into fewer addresses. This method can effectively save public addresses and reserve more private addresses for future customers. As shown in Figure 1.1, customer-facing NATs get their "public" IPv4 addresses from their parent CGNs' private IPv4 address pool. If a customer in the user network wants to visit the public Internet, his/her private address which will be translated to public IPv4 addresses via customer-facing NAT and CGN devices.

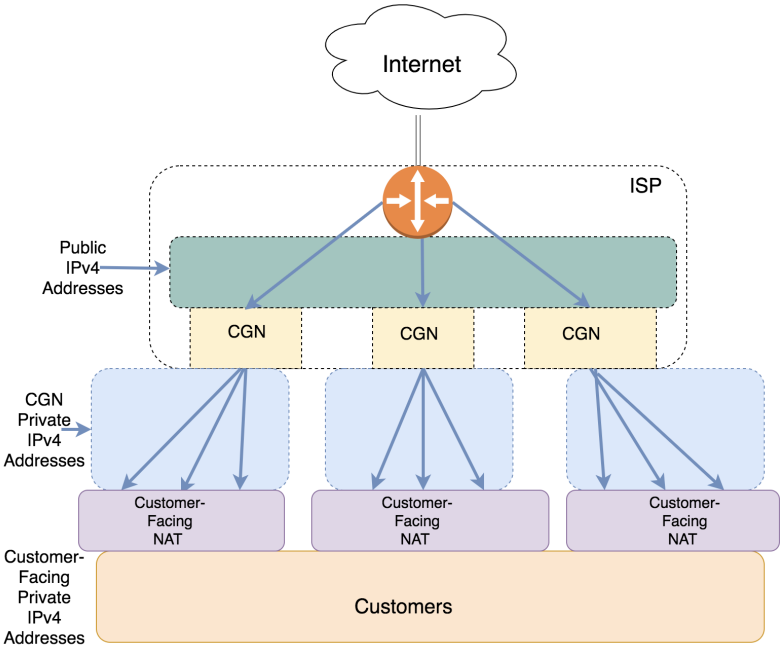

Figure 1.1: IPv4 Addresses Translation Network

The CGN in this project is deployed with FastClick. In order to measure the performance, basically, the latency and the throughput of a CGN, two Virtual Machines (VMs) are linked as shown in Figure 1.2. Those two VMs represent the generator/receiver and NAT respectively. The topology shown in Figure 1.2 is used for measurements in all tasks. The following are some terms that are shown in Figure 1.2 and used throughout the report:

- *Generator* is the machine which generates TCP and UDP flows;
- *Receiver* is the same machine as Generator, which receives testing flows and collects evaluation metrics;
- *NAT* is the machine which translates all the traffic received from *Generator* and passes them to *Receiver*. NAT can be run in two different modes: *single-core* NAT and *multi-threaded* NAT.

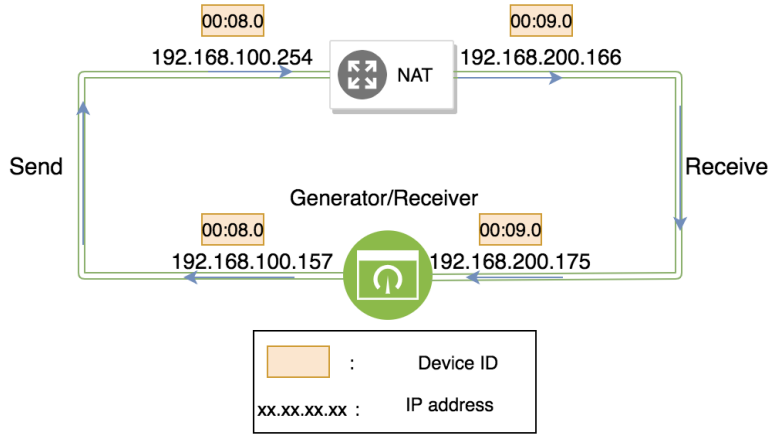

Figure 1.2: Simulation Network Topology

We briefly introduce the background of our project here. For further information, please see Chapter 2.

1.1 Problem description

CGN is one of the most effective ways to reserve public IPv4 addresses. As shown in Figure 1.1, the private IPv4 addresses behind it are used as public IPv4 addresses of customer-facing NATs, which indicates that there could be millions of hosts behind a CGN. However, if a CGN is built on a single-core host, it will suffer from high latency and bandwidth throughput limitations under heavy network transmission pressure[4]. It is inevitable to build multi-threaded NAT which can reduce the latency and improve throughput. The problem is how to implement a multi-threaded NAT and evaluate the performance of it. Although there are various methods to build a multi-threaded NAT, it is still important to compare those different methods and determine the most effective one.

1.2 Main Purpose

The project aims to construct a practical CGN on a multi-core host by using FastClick. Since FastClick is a modular router toolkit, a CGN built with it can be adjusted to customers' needs quickly. ISPs may get inspired by the final result of this project. If ISPs deploy CGN on a multi-core server, they will only need to adjust FastClick scripts according to their requirements instead of changing all hardware devices, which is much more sustainable and time-saving than traditional ways. Besides, the results of this project could be a hint for High-Speed

Network construction since it is a good example of realizing high-speed network function with FastClick module. Other thread-safe network functions could be developed based on the results of this project. Source code and configuration files are shared on our Github repository*, and besides, we will also explain the utilized multi-threaded algorithms and the usage of FastClick. We expect that our learning experience and final achievement will inspire and help people who are interested in High-Speed Network.

1.3 Goals

The ultimate goal of our study is to implement carrier-grade NAT on a single host with multiple CPU cores and use different techniques to ensure thread-safe and analyze how it performs. Based on that, we have several sub-goals during this project:

1. Building a single-core NAT and understanding its limitations
2. Implementing multi-threaded NAT in three different methods and determining how those methods improve the limited performance of single-core NAT
3. Doing literature study of related work and comparing some corresponding methods

We first deploy a single-core NAT by using FastClick on one host and profile the bottleneck of single-core NAT by measuring its latency and throughput. In October, we fully duplicate the NAT element per-core and achieve both software and hardware packets classification to realize multi-threaded NAT. Also, we deploy multi-threaded NAT by adding a spinlock to avoid conflicts among threads at the same time. In November, we give a rigorous review of several algorithms that could be used in multi-threaded NATs and realize the most functional one. Finally, we look back to all tasks and review how different algorithms affect the performance of CGN. You can find all the results and analysis in Chapter 5.

1.4 Research Methodology

This project is mainly classified as a quantitative analysis which conducts multiple experiments upon the multi-thread NAT features and gains a certain amount of data to draw the conclusion. The project aims at finding the reason that causes the performance limitation of single-core NAT during the deployment. We evaluate

* Repository link: https://gits-15.sys.kth.se/yfan/IK2200_NAT

traffic performance from two aspects, latency and throughput. Throughput metric plays a role where we could seek the maximum number of flows that causes the system to reach a limitation. The latency is measured to find the performance differences based on the increased number of flows.

By designing a single-core NAT system, we use traffic analyzer tools to evaluate the latency and throughput metrics, such as IPerf, Pktgen, etc. We perform our test 3-5 times to assure reliability of the measurements, otherwise, the result will not be representative nor persuasive. The research on packet generation will be summarized in the section Packet generator. During this project, confirmatory factor analysis, and correlation analysis are the critical part of our following study as profiling the performance.

Under constructive research in this project, we will scale the NAT in multiple cores. The methods to deploy contains finer-grained lock around the flow table and per-core duplication in software and hardware.

Finally, we will equip this project with the literature research on similar studies with functions like load-sharing, high-speed classification, etc. During the study of the related areas, we provide the reader with the limitation of each method, a summary of the methods and lessons learned report of our study.

1.5 Structure of this thesis

In this report, chapter 1 describes the problem and its context. All relevant background information about NAT and related tools is introduced in Chapter 2. Following this, chapter 3 describes the methodology and method used to solve the problem. The solutions and achievements are described in Chapter 4. The results are analyzed and evaluated in Chapter 5. Finally, Chapter 6 offers some conclusions and suggests future work.

Chapter 2

Background

This project is built on the idea of NAT described in RFC1631[1] and RFC2663[2]. A CGN, also known as a large-scale NAT (LSN) and described in RFC6888[5], is an extension of traditional NAT technologies designed for large-scale networks and Internet Service Providers (ISPs). CGN enables sharing of small pools of public addresses among many entities or hosts.

One method to build a CGN on a single host is to make full use of the multiple CPU cores. In this project, we mainly focus on the development of a multi-threaded NAT and make it fast enough in order to handle tremendous traffic flows.

2.1 Data Plane Development Kit

With the rapid emergence of cloud industry, infrastructure network is gradually biased toward the integration of architecture based on general-purpose computing platforms or modular computing platforms to support various network functions. High-performance programming is especially important in these service scenarios for massive data processing or mass users.

The Data Plane Development Kit[6] (DPDK) is designed to accelerate the packet processing. It provides library function and driver support for userspace efficient packet processing under x86 architecture and focuses on the high-performance packet processing in network applications. This will greatly enhance the ability of network devices to send and receive data packets.

DPDK application runs in the user-space and uses its own data plane library to send and receive data packets, bypassing the Linux kernel protocol stack for

packet processing. The Linux kernel runs DPDK application as a normal user-space process, including its compilation, linking, and loading methods, which are not different from ordinary programs. The basic steps include allocating the hugepage, loading the uio kernel module, binding the selected ethernet card and testing with sample DPDK application. In the next subsection, we will introduce the FastClick which is enabled with DPDK function to support the highest performance.

2.2 FastClick

FastClick[7] is an extended version of the Click Modular Router[8]. It defines many elements that can implement different routing functions. Besides, FastClick provides an interface which can connect these elements with simple and human-readable language (the Click language). In Chapter 4 we will describe how we realize NAT function by using FastClick module. And in this project, we will mainly use FastClick to configure our NAT scripts and finish throughput measurements.

2.3 Packet generator

There are several well-known packet generators we found such as Pktgen-DPDK, MoonGen, IPerf, Ostinato, Netperf and T-Rex. We conducted a study of all of these packet generators to compare their performances and chose the best one to use in our project. We installed all the software mentioned above and tested their packets generating and capturing performances.

2.3.1 Pktgen-DPDK

Pktgen-DPDK[9] is a traffic generator powered by the DPDK fast packet processing framework which can generate 10 Gbit traffic with 64-byte frames. This generator can be configured to start and stop flows at runtime and to display the real-time state of ports. We can use this function to detect our real-time NAT performance. It can work as a transmitter and a receiver at the same time and handle packets of Transmission Control Protocol (TCP), User Datagram Protocol (UDP), Address Resolution Protocol (ARP), Internet Control Message Protocol (ICMP), Multiprotocol Label Switching (MPLS), Generic Routing Encapsulation (GRE) and Queue-in-Queue. Pktgen-DPDK can generate packets by configuring source and destination IP, MAC, or ports. It also can be controlled over a TCP

connection.

In our case, we want to bind two ports with DPDK and implement Pktgen-DPDK on them. One port transmits packets to NAT and the other receives a packet from it. We can easily compare the difference between transmitting and receiving rates on the same screen.

Sadly, as we performed several tests using pktgen-DPDK, the receiving system doesn't show us the desired results as we envisioned. In our mind, the pktgen-DPDK behaves more likely a tool that will launch the DOS attack. Thus, even the packet is sent through DPDK, due to the complex network environment, it is really unclear where these packages will be lost. So in other words, using this tool to generates will increase our workload.

2.3.2 Moongen

Moongen[10] is a high-speed script packet generator. The entire load generator is controlled by Lua scripts: all packets sent are made by user-supplied scripts. Thanks to the fast LuaJIT VM and the packet processing library DPDK, it can saturate a 10 GBit Ethernet link with 64-byte packets while using only one CPU core. MoonGen can achieve this rate even if each packet is modified by a Lua script. It generates unique packets instead of replaying the same buffer.

MoonGen can also receive packets, such as checking packets dropped by the system under test. Since the reception is also completely under the control of the user's Lua script, it can be used to implement advanced test scripts. You can e.g. use two MoonGen instances to establish a connection with each other. This setting can be used to benchmark a middlebox such as a firewall. Figure 2.1 is the schematic diagram of the architecture.

The problem of MoonGen is really similar to that of Pktgen. Sometimes we cannot receive any packets. Besides, since we are experimenting on virtual machines, we have to use virtual drivers such as Virtio and VMXnet3. However, MoonGen does not support the ability to read and write timestamp on virtual drivers, which makes it impossible to use MoonGen for NAT performance testing.

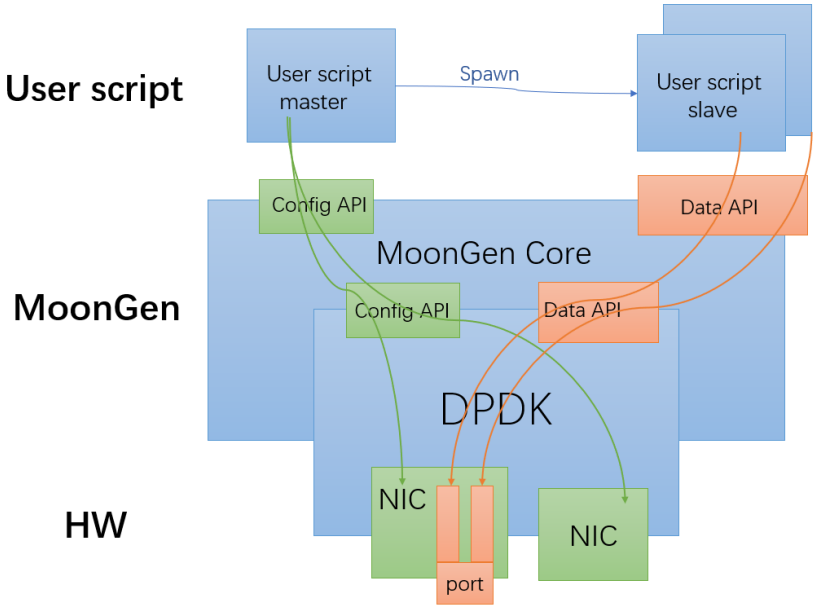

Figure 2.1: Architecture of MoonGen packet generator

2.3.3 Ostinato

Ostinato is an open-source, cross-platform packet/traffic generator and analyzer written in QT[11]. Ostinato employs an agent-controller architecture. It can send packets of several streams with different protocols at different rates. To deal with the combination of packages for various protocols, Ostinato has very flexible ways.

We use the GUI client with a comprehensive interface to perform our test. Ostinato performs smoothly in our test, but we cannot ignore the fact that the speed is too slow. Making it not an ideal traffic analysis tool to verify the NAT bottleneck effect.

Although it claimed that it supports DPDK, after we contacted its developers, they admitted that the DPDK plugin of Ostinato is no longer maintained and they decided to release the DPDK version of Ostinato next year. As a result, Ostinato is a wonderful tool but it won't be applied to our project.

2.3.4 Netperf

Netperf[12] is a benchmark used to measure the performance of lots of types of network including TCP, UDP, and SCTP for both IPv4 and IPv6. Netperf works in

client-server mode. The application of Netperf is to measure both unidirectional throughput and end-to-end latency in a configured measure time.

Although Netperf is easy to install and operate, it hasn't supported DPDK so far and it will no longer be updated.

2.3.5 IPerf3

The iperf series of tools perform active measurements to determine the maximum achievable bandwidth on IP networks. It supports tuning of various parameters related to timing, protocols, and buffers. For each test, it reports the measured throughput, loss, and other parameters.

The latest version sometimes referred to as iperf3, is a redesign of an original version developed at NLANR / DAST. Iperf3 is a new implementation from scratch, with the goal of a smaller, simpler code base, and a library version of the functionality that can be used in other programs. Iperf3 also incorporates a number of features found in other tools such as nuttcp and netperf but missing from the original iperf. These include, for example, a zero-copy mode and optional JSON output. However, iperf3 is not backward compatible with the original iperf.

However, due to the poor compatibility and stability problem of IPerf3 with DPDK implemeted, we have not done more in-depth research on it.

2.3.6 T-Rex

While conducting our tests, we realized that T-Rex was the most suitable tool for our purposes. T-Rex[13] is developed based on DPDK and it is an open source tool for packet generation and distribution, which runs on standard inter-processing chips. This tool supports stateful and stateless modes. The stateful mode can describe application scenarios from Level 4 (L4) to Level 7 (L7), while the stateless mode is mainly used to form customized packets. Now, latest version T-Rex supports more features such as multi-streaming, changing package fields, and supporting statistics summary, latencies, and jitter information based on each stream.

In our experiment, we need to test packets of multiple protocols under different sending rates and record the latency data and total throughput. To fulfill this requirement, the stateless mode of T-Rex is sufficient. The official document introduced us to the T-Rex stateless GUI which can highlight the experiment with

friendly interfaces and monitor the states of each port conveniently.

Advantages of T-Rex over other packet generators:

- Customized flows can be generated.
- T-Rex is more powerful for reaching line-rate throughput.
- Mixed UDP and TCP flows can be created.
- High-performance full line rate.
- Generate both sides of the traffic clients and servers.

2.4 Profiling tools

2.4.1 Perf

Perf[14] is a Linux profiling tool that commonly used for searching performance bottlenecks and location of the hot code. It can interrupt CPUs in fixed sampling time and record which process ID (PID) or function is running. The Linux Perf tool has many functions in system profiling, for example, the perf top command can view the real-time CPU utilization based on a specified performance event (default is CPU cycle), and display the most expensive function or instruction. We can sort the list by different software/hardware events or even process ID. After recording sufficient data, perf report command reads the data file created by perf record and gives out the hotspot analysis result. In Task I, we need to use Perf tool to prove that the bottleneck of our topology caused by single-core NAT.

2.4.2 Flame Graphs

As we researched about system profiling, we found an interesting tool named Flame Graphs[15] which can produce the perf result in a graph looks like a flame, showing the call stack of CPUs. It is a Scalable Vector Graphics (SVG) format graph that allows you to click/fly on each bar of the flame in order to get more detailed information.

To explain the concept of Flame Graph briefly, the following figure 2.2 gives an example of a flame graph, it can be zoomed in by clicking the link of the graph.

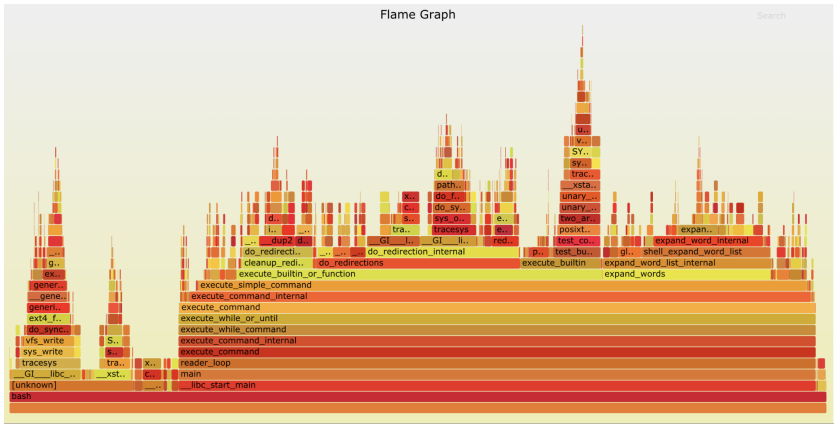

Figure 2.2: An Example of Flame Graph

Each box represents a function. The deeper the call stack, the higher the flame. The top is the function being executed, and the lower is its parent function. In other words, the height of a Frame Graph is proportional to the call stack depth. The width of each box represents the number of samples. If a function occupies a wider width on the x-axis, it means that it is drawn more times, on the other hand, it represent that a process needs longer time to execute. Note that the x-axis does not represent time, but all call stacks are merged and arranged in alphabetical order.

The reason to use this flame graph in our profiling stage is that we can see which function of the layer occupies the largest width, once it reaches the “plateaus”, the function may have performance problems.

Chapter 3

Methodologies and Methods

The purpose of this chapter is to provide an overview of the research methods used in this project. In this chapter, section 3.1 describes the research process and details of the research paradigm. Section 3.2 focuses on the data collection techniques used for this research. Section 3.3 describes the experimental design. We explain the techniques used to evaluate the reliability and validity of the data collected in Section 3.4. Section 3.5 describes the method used for data analysis and section 3.6 explains the evaluation framework

3.1 Research Process

Figure 3.1 shows the steps conducted in order to carry out this research. In background material research, building FastClick testbed and single-core NAT measurement parts, we work together. In other parts, we are divided into two groups and work in parallel.

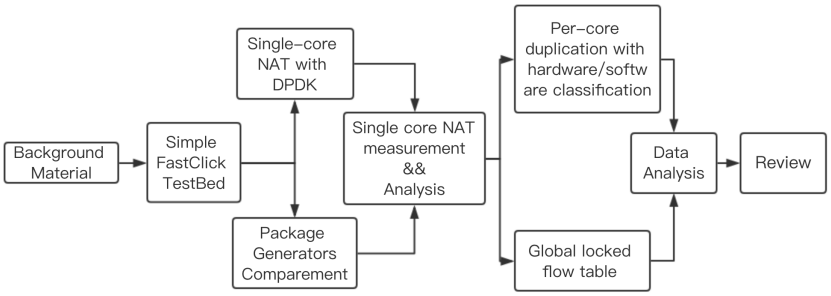

Figure 3.1: Schematic Diagram of Research Process

3.2 Data Collection

The main purpose of this project is to build a CGN for realistic usage. CGNs can be regarded as central stations for information traffic. These data passing through may include private or confidential data, which make people concerned. In this project, we plan to build a CGN on a VM and generate data from T-Rex or FastClick. As Figure 1.2 shows, the Generator/Receiver connects to the NAT directly on both ports. Thus, no data will flow out of this system and no foreign data will be captured by T-Rex. Thus, the data that we collect and analyze will not influence people's normal life or leak their privacy. The procedure of this project follows the IEEE Code of Ethics[16] all the time.

3.2.1 Data Sampling

All tasks in this project aim to collect the latency and the throughput value. The latency sample is the latency value collected in one flow randomly chosen from flows that is sent out by the T-rex traffic generator. The latency sample flow can represent all flows because the CGN handles all flows in the same way.

The throughput value is measured by FastClick. FastClick sends out as much traffic as possible and detects the amount of traffic that CGN allows to pass through. Due to the throughput measurement strategy, all throughput results are recorded and no sampling is needed.

3.2.2 Sample Size

In all tasks, the latency sample is collected by T-rex in 100s. One latency sample contains 100 latency measurement results. The throughput sample size is 20. Each throughput sample contains the TX rate and the RX rate of FastClick. The Units of measurement results are listed in Table 3.1

Parameter	Unit
Average Latency	μs
Transmit Packets per Second(TX)	Mb/s
Receive Packets per Second(RX)	Mb/s

Table 3.1: Sample Units

3.2.3 Testbed Environment

In this project, two 4-core VMs are deployed, one for evaluation and another running as the CGN, named as NAT. These two VMs are connected as Figure 1.2. Two Ethernet ports are put and assigned IP addresses on both NAT and the evaluation machine, which is called Generator/Receiver. On both VMs, DPDK environment is setup to meet High-Speed Network requirement. FastClick is installed on both machines while T-rex is installed only on Generator/Receiver. T-rex runs on Generator/Receiver and FastClick runs on NAT when it's time to measure packets delay caused by NAT. FastClick runs on both VMs to measure the throughput of NAT.

3.3 Planned Measurements

The planned measurements of different tasks are introduced separately.
1. Task I aims to build a single-core NAT and profile its limitations. Thus, the measurements ought to prove that single-core NAT works properly and a bottleneck exists.

- Two VMs are connected and setup as section 3.2.3.
- Run single-core NAT configuration written in FastClick on NAT and run stateless T-rex on Generator/Receiver.
- Ensure that T-rex could receive packets passing through NAT.
- Use Iperf and tcpdump to check if NAT changes the IP headers of packets.
- Measure the latency caused by NAT that only one flow passing through. Then stepwise increase the number of flows by 500 and repeat such measurement until there are 20k flows in total.
- Stop T-rex on Generator/Receiver and run FastClick instead. Measure the throughput of single-core NAT with FastClick. Start with a single flow and increase the number of flows by 10k until there are 100k flows in total.
- Run DPDK-forwarding configuration on NAT machine. The only difference between DPDK-forwarding configuration and single-core NAT configuration is that DPDK-forwarding configuration sends packets to Generator/Receiver without changing IP headers.
- Measure the latency and throughput of DPDK-forwarding in the same way as single-core NAT.

- Use Perf to profile what causes the bottleneck performance.

2. The main purpose of task II is to complete several multi-threaded NAT and compare their performances.

- Run 2-core configuration on NAT. 2-core configuration follows the same idea of single-core NAT configuration but deploys two cores to handle the traffic. Use Htop to check if both cores work. Then, measure the latency and throughput of 2-core NAT.
- Run spinlock configuration on NAT. 2-core configuration follows the same idea of single-core NAT configuration but deploys two cores and four cores to handle the traffic. Measure the latency and throughput of 2-core and 4-core NAT.
- Run hardware-classification NAT configuration on NAT. hardware-classification NAT configuration follows the same idea of 2-core NAT configuration but deploys one more core to classify the packets.

3. Task III requires literature review of related paper. Based on the review, the multi-threaded NAT should be updated to improve its network performance.

- Compare the lookup algorithms mentioned in the related paper and presents the advantages and disadvantage of every algorithm.
- Choose one algorithm and implement it with C++ in FastClick. Measure the new NAT performance and compare it with the NAT without the algorithm.
- Based on the results of previous profiling, improve the source code of NAT implementation in FastClick.

3.4 Reliability and Validity of the Data Collected

3.4.1 Reliability

As mentioned in section above, the collected data is the throughput and latency of CGN working in different CPU allocation modes. All data was collected from the Generator/Receiver machine. In order to be sure that these results are reliable, we repeated each latency measurement for 100 times and each throughput measurement for 20 times. Repeated measurements could effectively reduce the noise and distinguish the data. Besides, in order to avoid artificial error, 3 group members repeated the all measurements and got similar results.

3.4.2 Validity

The target data of this project is consist of latency and throughput. To check the validity of collected result, we checked the IP headers of packets to ensure that the packets Generator/Receiver got have passed through the NAT machine. Also, we measured the latency and throughput of packets when two Ethernets of Generator/Receiver were connected directly. The result of the direct connection showed that the latency and throughput were constant when flow number changed.

Besides, we deployed DPDK-forwarding configuration as a contrast. The result of DPDK-forwarding proved that the data collected in this project was valid.

3.5 Planned Data Analysis

Because the latency results and the throughput results were collected in different ways, they ought to be processed differently.

3.5.1 Data Analysis Technique

Latency measurement results were collected from the TUI panel of T-Rex on Generator/Receiver machine. The latency value on the panel was the same as the previous one if Generator/Receiver didn't receive the packets with the timestamp. Thus, those repeated results should be filtered out in the data processing section. After that, the processed data was stored in CSV-formatted files and mapped to ErrorBar Plots.

Throughput measurement results were collected from FastClick on Generator /Receiver machine. FastClick presented measurement results of NAT directly and we didn't need to process the collected data anymore. The Python script drew the ErrorBar Plots of throughput for further analysis.

3.5.2 Software Tools

- Pycharm
  PyCharm is an IDE for Python. All Python scripts in this project is written in Python 2.7 and PyCharm is very compatible across all Python edition.

- Microsoft Excel
  We stored all data in CSV-formatted files. Microsoft Excel is a widespread software which has a user-friendly interface and we can look into the data easily once something is wrong.

3.6 Evaluation Framework

- Iperf & Tcpdump
  Use Iperf and tcpdump to evaluate the flows going through NAT. Iperf is a tool that actively measures the maximum achievable bandwidth on IP networks and tcpdump is a common packets analyzer running under command line, which will give us all the information of transmitted and received data packets. Combining these two tools makes it possible for us looking into IP headers of packets. This can verify whether a packet has been successfully rewritten by middle NAT machine.

- Perf
  Use Perf running on the NAT machine to find out if the CPU utilization of FastClick functions changes with and without NAT. The environment and the network configuration are the same as ordinary measurements.

Chapter 4

Milestone Achievement

In this Chapter, we will introduce the milestone achievements. In section 4.1, we shows how the single-core NAT works and how it is implemented by using FastClick elements. Section 4.2 introduces several methods to build multi-core NAT. In section 4.3, we will discuss how NAT code is implemented in the FastClcik. Section 4.4 gives out a series of methods which can improve the performance of NAT.

4.1 Single-core NAT Implementation with FastClick

The configuration of single-core NAT is based on Click language and compiled by FastClick. In order to implement NAT, we need to realize functions by using Click Elements. The following figure 4.1 is a flowchart to show how we use element configurations to finish the NAT process.

First, when a NAT machine receives a packet, it needs to distinguish the type of the packet. If the packet is an ARP request, NAT will generate a response and send it to the corresponding interface. If the packet is an ARP response, then NAT will encapsulate ethernet header, which is found via ARP, into IP packets. If the packet is an IP packet, NAT will classify its type (TCP, UDP or ICMP) and send the packet into corresponding Rewriter Elements.

The procedures for the different rewriter elements are really similar. The elements used in single-core NAT are TCPRewriter, UDPRewriter, ICMPRewriter, and ICMPPingRewriter elements. The parameters which specify the new source IP addresses and source ports in these Rewriter Elements is from IPRewriterPatterns. After all of these finished, NAT will broadcast a ARP query to request the Ethernet address of destination machine and send out the rewritten packet.

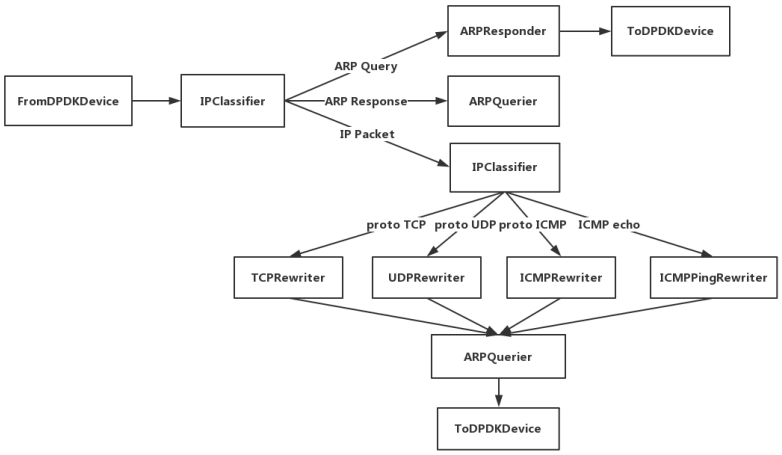

Figure 4.1: Flowchart of single-core NAT configuration

4.2 Multi-core NAT Implementation

4.2.1 Global Locked Flow Table

When it comes to multi-core NAT, a problem arises that if let a multi-threaded process visit a single flow table without any protection, this may cause huge conflicts. For example, if thread A is searching for a flow in a flow table while thread B is updating flow table, then the searching result of A will mismatch the current flow table. This situation will get worse when the number of threads increases.

In order to solve this problem, we used a spinlock to avoid conflicts of concurrently processing the same resources. A spinlock can only be held by an executable thread at most. If an execution thread tries to get a spinlock that has been held (contested), it will keep busy looping-spinning-waiting for the lock to be reused. If the lock is not contested, the execution thread, which is requesting the lock, can get lock acquired immediately and continue executing. At any time, spinlock prevents more than one thread of execution from entering the critical region at the same time.

In FastClick, one of the methods is that we can change our script from single-core NAT to a multi-threaded version by directly applying spinlock element to

FastClick. The structure of this method is shown below:

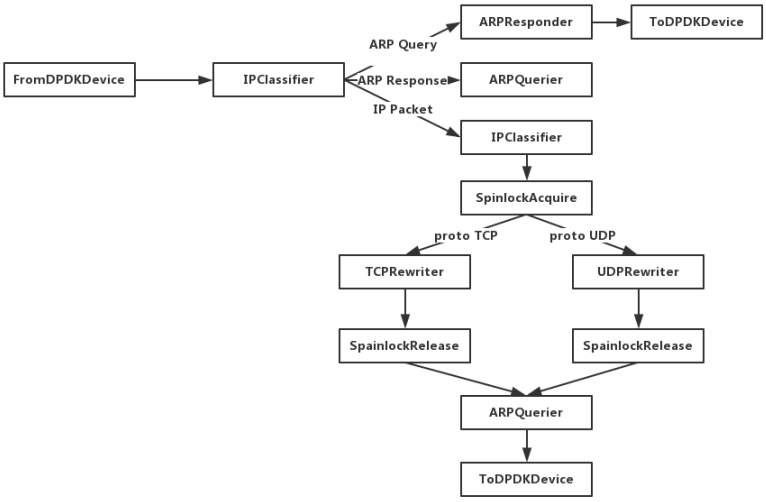

Figure 4.2: Flowchart of multi-threaded NAT configuration with Locked Flow Table

As seen in figure 4.2, there are SpinlockAcquire and SpinlockRelease elements before and after TCPRewriter and UDPRewriter elements. SpinlockAcquire will call the global spinlock method which can lock up the corresponding flow table until CPU finish the operation. Therefore, only one thread at a time can handle Rewriter elements.

But if we only simply use lockaquire and release elements, this rough lock will lock up all the rewriting operation. This also includes some unnecessary functions, which will greatly slow down the acquisition of locks by other CPUs. In order to avoid this defect, we introduced a finer-grained lock.

The following figure 4.3 is roughly based on file udprewriter.cc and shows how a UDP packet goes through the UDP Rewriter element. Since there are several functions in class UDPRewriter extended from its parent IPRewriterBase, the flowchart also contains some functions in iprewriterbase.cc.

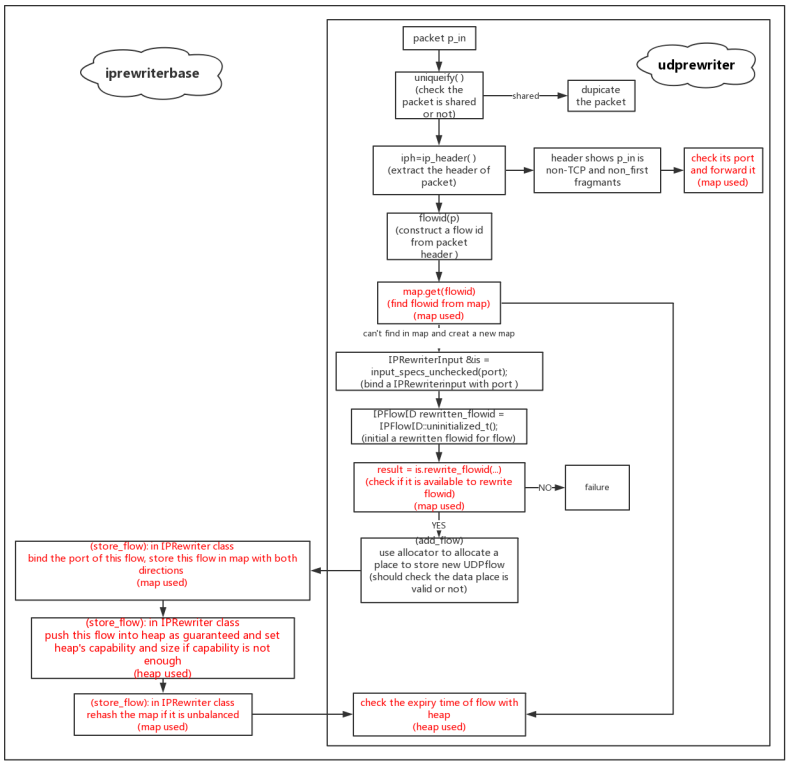

Figure 4.3: Process of UDPRewriter element handling UDP packet

The functions that access flow table are what we are interested in, for we should use a spinlock to avoid several threads handle it at the same time and cause errors. As you can see in the figure above, the red font part is the function that we find relative to the flow table. In detail, the flow table includes two types:

- First is called map which is used to store flowid, source and destination addresses and source and destination ports.
- Second is called heap which is used to store each flow's expiration time, rewriter element will delete it both in map and heap.

When heaps or maps are being processed, we want to make sure that no other threads can process them, which will prevent conflicts among threads.

After finding all parts related to heaps and maps, locks are added around them. First, we declared a protected or public Spinlock type variable named "_lock" in class IPRewriterBase. By declaring "_lock", we can acquire and release a spinlock

at any place in all child classes of IPRewriterBase among different threads. Then we used Spinlock.acquire() and Spinlock.release() functions to acquire and release the locks around all parts related to heaps and maps. When a thread acquires a lock, it should first check if this part is blocked by other thread or not. Finally, we finished finer-grained spinlock by testing its latency and throughput. In Chapter 5, measurement results and discussions will be presented.

4.2.2 Per-core duplication, with software classification

It is really similar to the single-core NAT, but what we need to do is to use a CPUSwitch element in order to duplicate the process of NAT function. When a packet comes back, we need to send it to a corresponding Rewrite flow table based on its destination port. The following figure 4.4 shows a flow chart while using multi-core NAT.

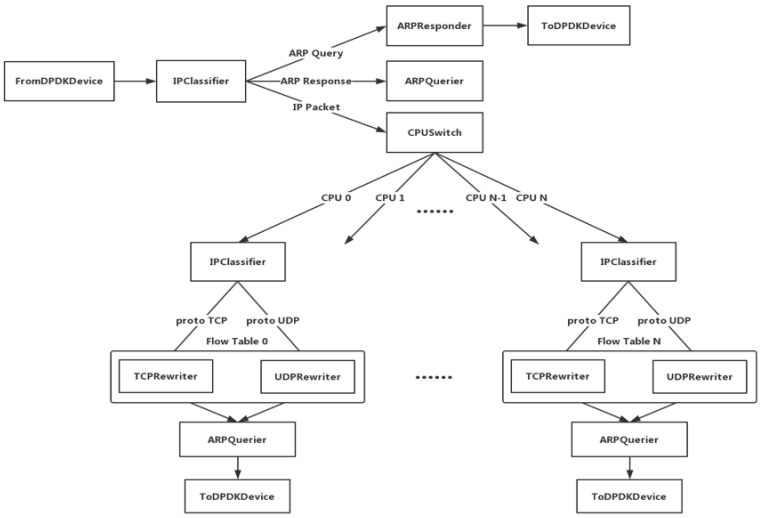

Figure 4.4: Flowchart of Per-core Duplication with software classification

For each Rewriter Element, they maintain different flow tables. For example, TCP and UDP Rewriter under CPU 0 maintain flow table 0. After choosing the corresponding CPU, the packets will be rewritten and sent to the corresponding ToDPDKDevice Element. And while dealing with return ACK packets, the following figure 4.5 shows how a NAT will work.

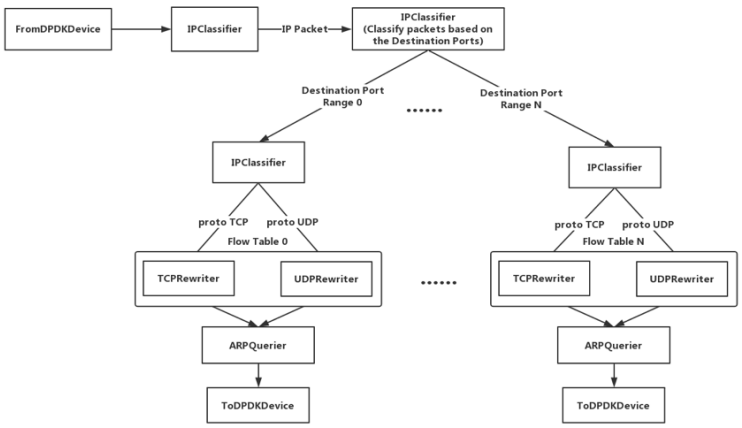

Figure 4.5: Flowchart of Returning ACK

When an ACK comes back from the destination terminal, NAT will classify the destination port and push the packet to the corresponding flow table. For example, packets pushed to CPU 0 will be rewritten the source port within range 5000-5500, then the returning ACKs within destination port range 5000-5500 will be pushed to flow table 0, which maintain the mapping to the original packet.

4.2.3 Per-core duplication, with hardware classification

The idea of hardware classification is identical to the software classification. The only difference is how to deal with those ACKs. In hardware classification, a separate core will be assigned to FromDPDKdevice and IPClassifier elements to complete packets classification and receiving. The following figure 4.6 shows how this works.

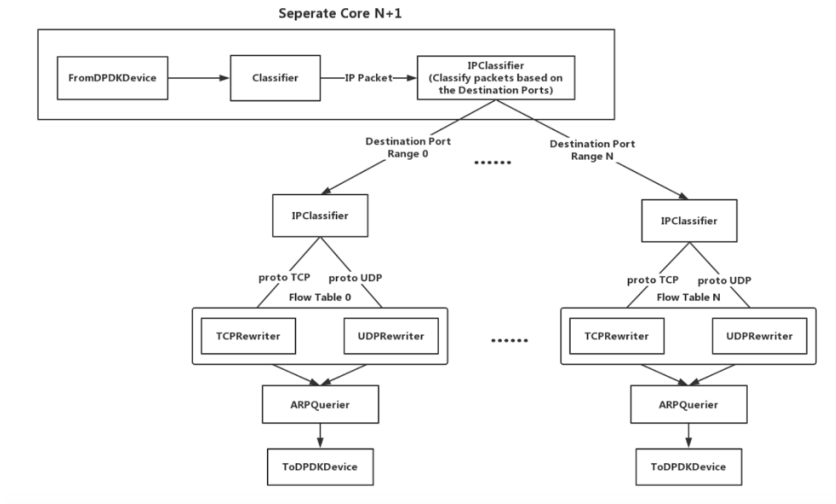

Figure 4.6: Flowchart of Returning ACKs while using Hardware Classification

After achieving the incoming packet, the separate core will check its packet type and destination port, then send it to the corresponding flow table which stores the mapping information of it.

4.3 NAT Code implemented in FastClick

NAT code implemented in FastClick mainly includes two parts: Hashtable and heap tree.

Hashtable in FastClick uses chaining as its collision resolution strategy. In the link method, all flows that are hashed into the same slot are placed in a bucket list. The following figure 4.7 shows a typical chaining hash.

The hashtable in the figure above contains eight buckets, the position of the top-down yellow background. If a new element is added to the hashtable, it will be added to the bucket corresponding to the hash of its key. In our case, a key is a flowID. If an element already exists in the same location, the new element will be added to the front of the list.

Besides, since NAT should deal with a huge amount of flow and in order to avoid huge waste of the memory space, Click rebalances the hashtable by doubling the number of buckets. If the number of flows is twice larger than the number of bucket, Click will double the number of buckets and rehash all the flows, which

can greatly avoid the collision and reduce the element searching time.

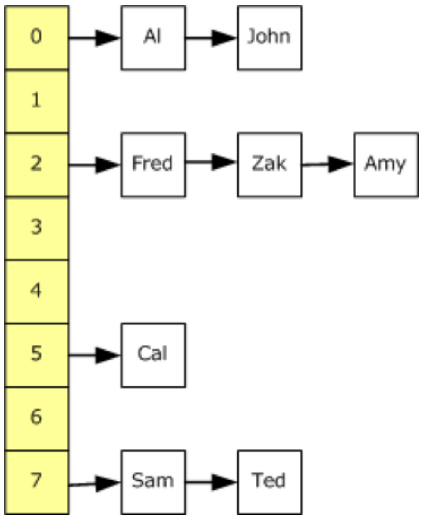

Figure 4.7: Typical Example of a Chaining HashTable

Another important component is the heap tree, which is used to deal with flow timeout. The following figure 4.8 shows a typical heap tree.

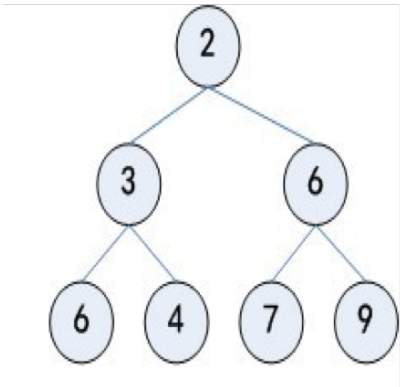

Figure 4.8: Typical Example of a Small-Root Heap Tree

A heap is also called a priority queue. The operations allowed in the queue are FIFO, inserting elements at the end of the queue and extracting elements at the head of the queue. The same is true for heaps, where elements are inserted at the bottom of the heap and taken out at the top of the heap, but elements in the heap are arranged in a certain order of priority rather than in the order of arrival. This

priority can be the size of the element or other rules. As shown in the figure above, the heap is the small-root heap. The heap priority is that the small elements are in the first place and the large elements are in the second place. The resulting heap is called the smallest heap. The top element of the smallest heap is the smallest in the whole heap, and each branch can also be considered as a small-root heap.

The heap tree in Click is also a small-root tree. For every flow that needs to be translated, NAT will give it a timestamp and store into the heap tree. First, the flow will be attached as a leaf and then rearrange the position in order to become a heap tree by calling the change_heap function. While checking the expiry, heap tree will compare and pop out the root point and destroy the corresponding flow, then call the change_heap function to reorganize the order of remaining flows.

But the heap in FastClick is very hard to make thread-safe and not so efficient. We will discuss the performance in Chapter 5. Besides, the hashtable is not multi-thread safe. In FastClick, based on the number of the CPU cores, a corresponding number of hashtables will be created and assigned to each core. Cores will maintain each hashtable separately without disturbing others. The following section 4.4 introduces the method from literature review which can help us overcome those problems and improve NAT performance.

4.4 Literature Review

To start with the research of scaling NAT using multiple CPU cores, we did a thorough search from an algorithmic perspective that may help with the understanding. Before we located the main technology that can do the load-sharing work, we found the effective way is to search for the resources with keywords like “high-speed”, “scaling” and “packet classification”.

After reading through the papers we’ve found. The main findings are summarized into 2 different table A.1 and A.2 that can be found in the appendix. Among our findings, there are several types of algorithms may enhance the performance of the lookup and update module in NAT.

4.4.1 Packet Classification Algorithm

Packet classification[17] is simply describing the process of categorizing the packets according to different rules into “flows” through the network. These rules are typically a field or a combination of certain fields at the beginning of the

packet.

First, it performed a protocol parsing and field extraction operations on the data packet, and then the data packet is classified according to the protocol type and the classification field. In most cases, the packet classification is based on quintuple or any other combination of quintuple, they are destination IP (32 bits), source IP (32 bits), destination port (16 bits), source port (16 bits) and protocol type(8 bits).

We evaluated these algorithms based on it time complexity and the difficulty of achievability in terms of update time. From the paper and resources we've found, most of them don't provide with the incremental update, and to do the update in the global mode will occupy a huge amount of resources. So most of them had poor performance on update speed and won't be a good choice to implement in the next stage.

4.4.2 Routing Lookup Algroithm

After the summarize the findings in packet classification algorithm, we turn our target into study the faster, more stable and up-to-date lookup algorithm in routing function. The following figure 4.9 and 4.10 we plot is based on table A.2, which represent the theoretical time complexity and update complexity of the algorithm we've found.

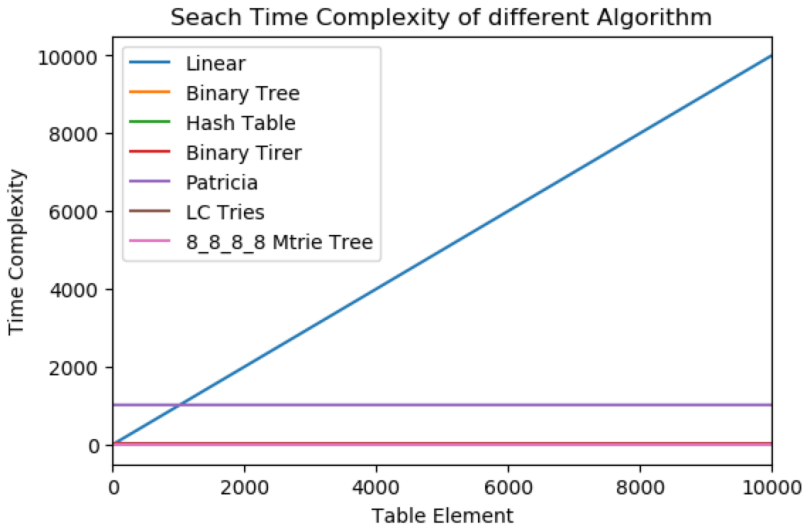

Figure 4.9: Lookup time complexity of different algorithms

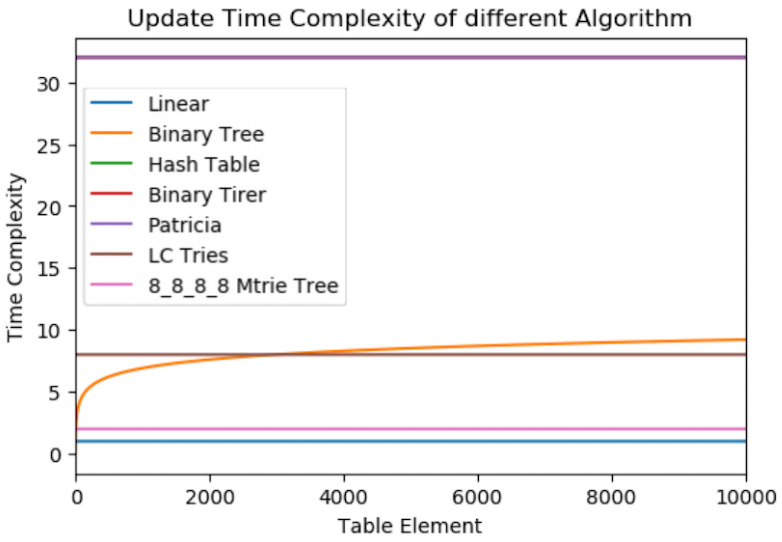

Figure 4.10: Update time complexity of different algorithms

4.4.3 Bihash

Bihash(Bounded-index extension hash) is the hash function that used in VPP (Vector Packet Processing)[18]. The specifications of Bihash can be included as follows:

- Bihash supports different sizes of hash keys and values including 8 bits, 16 bits and other sizes less than 64 bits.
- Bihash uses 64 bits hash and supports double-layer hash search, the first layer is bucket search and the second layer is page search.
- The size of buckets and pages is power of two, so that only bitwise and operation is needed in hash search.
- Decrease the fragmentation of heap by using free lists and increase the efficiency of distribution.

Key is the target of hash in Bihash calculation process. To avoid GCC bug and increase the efficiency of calculation, the length of key should be multiple of 8 bits. The result of hash is 64 bits, and the structure of hash is shown in figure 4.11 below.

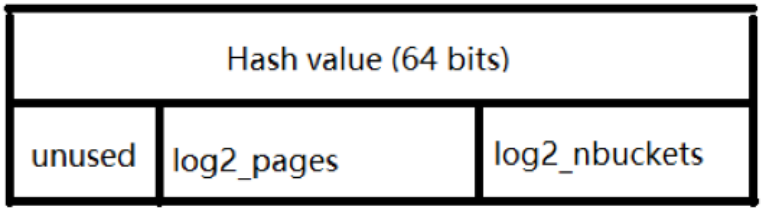

Figure 4.11: Hash structure

Here, we will briefly introduce bihash's realization process including init, add, del, search and free. After init process, the memory condition is shown as figure 4.12 below

Figure 4.12: Initialized memory

When it comes to add buckets, the memory structure will be different between adding empty bucket and adding non-empty bucket. The difference is shown in figure 4.13 and 4.14.

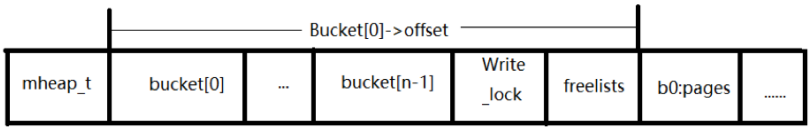

Figure 4.13: Memory after adding empty bucket

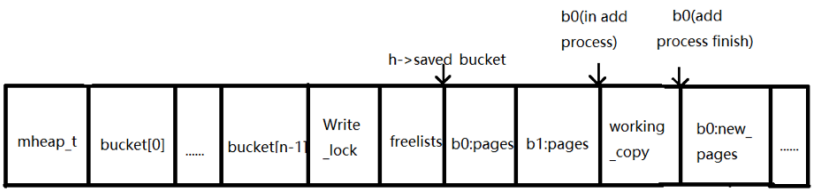

Figure 4.14: Memory after add non-empty bucket

The realization of the operation research is that: hash the key and find the bucket according to hash value, return 0 if it is found. Then find it in b0:new_pages by hash or linear search. Update $bucket[0]->cache$ if it is found, or return -1.

4.4.4 Cuckoo Hash

Cuckoo Hashing is a technique for resolving collisions in hash tables that produces a dictionary with constant-time worst-case lookup and deletion operations as well as amortized constant-time insertion operations.
It represents a simple hash table where:

- Lookups are worst-case O(1);
- Deletions are worst-case O(1);
- Insertions are amortized, expected O(1);
- Insertions are amortized O(1) with reasonably high probability;

There are two hash tables T_1 and T_2 each with r buckets with independent hash functions h1 and h2 each mapping a universe U to bucket locations $\{0,...,r-1\}$. A key x can be stored in exactly one of the locations $T_1[h_1(x)]$ and $T_2[h_2(x)]$. A lookup operation in Cuckoo Hashing examines both locations $T_1[h_1(x)]$ and $T_2[h_2(x)]$ and succeeds if the key x is stored in either location. Formally, then, the following algorithm 4.1 describes the lookup behavior of Cuckoo Hashing.

$function \quad lookup(x)$

$$return T_1[h_1(x)] = x \cup T_2[h_2(x)] = x \tag{4.1}$$

end

The cuckoo hash is multi-thread safe and decreases the lookup operations significantly. The idea is to use two hash functions instead of one, so that there are two buckets for the key to reside in. If all slots in both buckets are full, the key is inserted at a random slot within the two buckets, and the existing key at the randomly chosen slot is evicted and reinserted at another bucket. Figure 4.15 below shows the process of data packets in Cuckoo Hash.

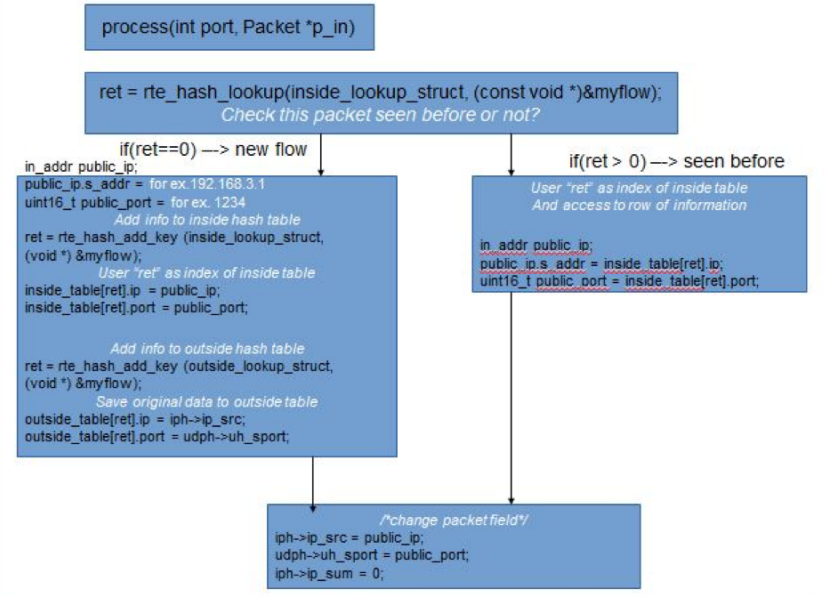

Figure 4.15: Packet Processing in Cuckoo Hash

Cuckoo hash implementation pushes elements out of their bucket, if there is a new entry to be added which primary location coincides with their current bucket, being pushed to their alternative location. Therefore, as user adds more entries to the hashtable, distribution of the hash values in the buckets will change, being most of them in their primary location and a few in their secondary location, which the later will increase, as table gets busier. This information is quite useful, as performance may be lower as more entries are evicted to their secondary location. You can find the code which implements the Cuckoo Hash in our Github Repository*.

* Repository link: https://gits-15.sys.kth.se/yfan/IK2200_NAT

Chapter 5

Results and Analysis

This chapter aims to present the data collection and analysis. In section 5.1, parameters which are set during the whole test are stated again. Section 5.2 presents the data and corresponding analysis of Task I. Section 5.3 describes the results collected from the multi-threaded NAT network with different strategies. Section 5.4 introduce a new mechanism and gives out the performance comparison. In section 5.5 and 5.6, we will describe the reliability and validity of the collected data. Finally, in section 5.7, we will make a discussion based on all the results we got from the previous section.

5.1 Testbed Parameter Settings

To ensure the reliability of data analysis, we used exactly the same testbed settings in all tasks.

The generator of latency measurement, T-rex, sent total 20K packets per second(pps) with multiple flows. The flow number changed in different tests but the total transmission(TX) rate was a constant. Besides, since T-Rex could measure accurate latency only if its CPU usage was lower than 80%, the chosen TX rate was the highest rate that T-Rex could afford. To profile the relationship between the latency and the flow number, for each latency test, the number of flows has been chosen at 500 intervals from 1 to 20k.

The generator for throughput measurements, FastClick, was written with click configuration. FastClick could produce high TX rate up to 36M pps when it was set to full-CPU utilization. The maximum number of flows it could generate was 100k.

5.2 Task I Results and Analysis

Task I aims to build a single-core NAT and identify its bottleneck. In order to find out the bottleneck of single-core NAT, its performance was evaluated in terms of latency and throughput.

5.2.1 Latency of single-core NAT

Based on the testbed we have mentioned above, the following figure 5.1 gives out the UDP traffic latency of the NAT machine when it run a simply-forwarding and single-core NAT script.

The latency value of UDP traffic had a regular pattern. As shown in the figure, there were five peaks in total which appeared at 1000 flows, 3000 flows, 4500 flows, 8000 flows, and 17000 flows respectively. Each number of flows that caused a latency peak was roughly twice as large as the previous one. For example, point 3000 was nearly the double of 1000. Besides, the latency value of each peak point was also doubled.

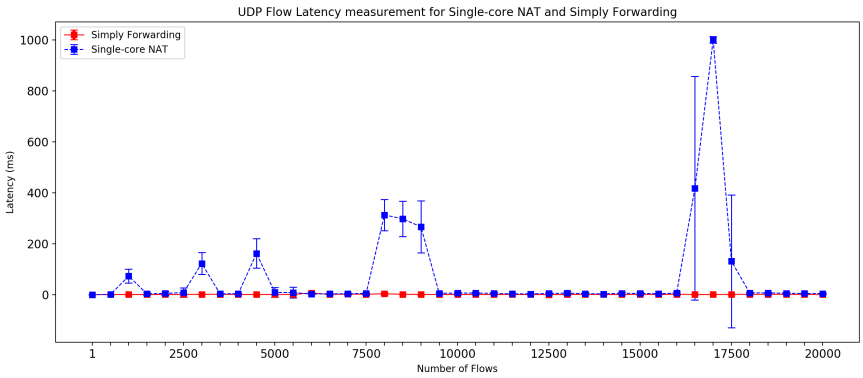

Figure 5.1: Latency Measurement of UDP Traffic while using single-core NAT and simply forwarding configuration

Figure 5.2 shows all the values while the y-axis was limited within the range of 0-10000 us, which could show a better overview of how latency changes.

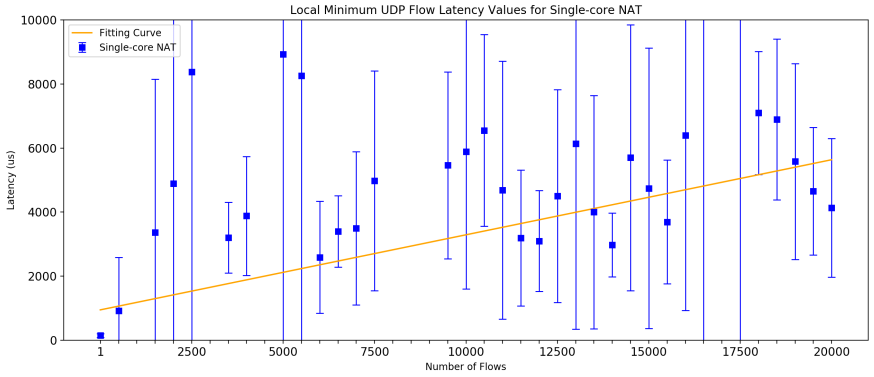

Figure 5.2: Local Values of UDP Traffic Latency Measurement plot

After excluding the extreme values, the latency values of UDP traffic increased with the number of flows. The orange fitting curve drawn in this figure showed a linear growth trend of minimum latency value after the appearance of peak latency values.

Shown in Figure 5.3 and Figure 5.4, the latency measurement result of TCP traffic was pretty similar with the result of UDP traffic: There were 5 peaks; The locations and values of those peaks had particular rules; The latency of TCP traffic also had linear increasing trend as the number of flows increased.

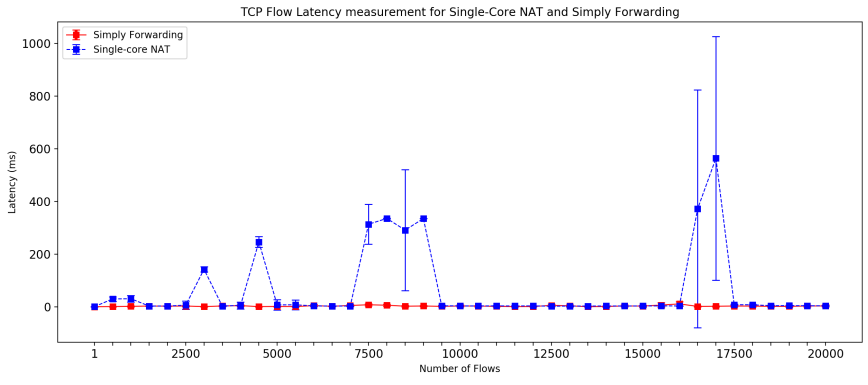

Figure 5.3: Latency Measurement of TCP Traffic while using single-core NAT and simply forwarding configuration

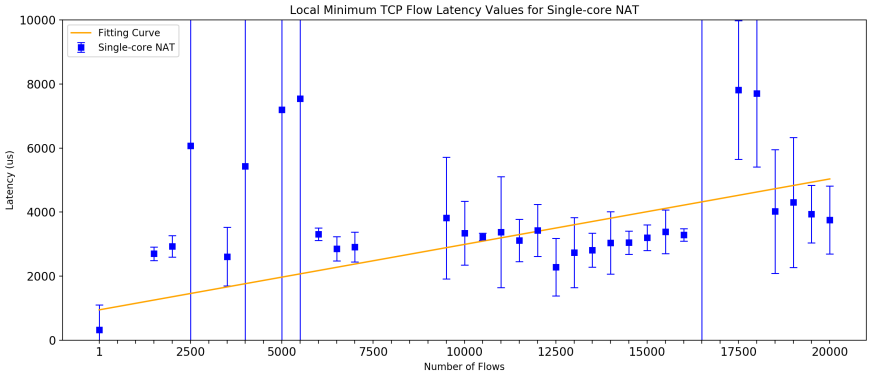

Figure 5.4: Local Values of TCP Traffic Latency Measurement plot

The difference of latency results between TCP traffic and UDP traffic was that the normal latency values of TCP was more stable while UDP traffic had huge latency range during all the tests. Because of the similarity of TCP and UDP results, the rest experiments focused on results of the UDP traffic only.

5.2.2 Throughput of single-core NAT

Throughput was another import metrics for us to evaluate the NAT performance. Figure 5.5 showed how the throughput changed with the increasing number of flows.

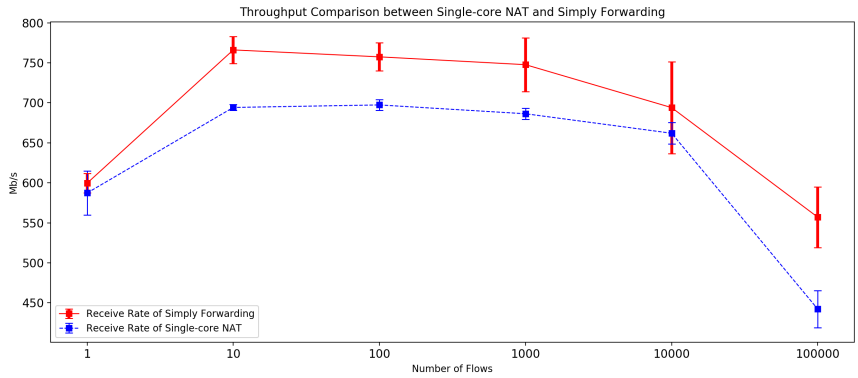

Figure 5.5: Throughput measurement results of single-core NAT and simply forwarding with exponentially increased number of flows

In the figure, the red line represented the throughput of the NAT machine with simply forwarding script and the blue line represented the throughput of the NAT machine running single-core NAT script.

Both lines increased when the number of flows increased from 1 to 10 but the red line increased more rapidly. The increase was caused by FastClick: the TX increased as the number of flows increased 1 to 10 and kept stable when there were more than 10 flows.

There was a dramatic decrease between 10000 flow and 100000 flow in both lines because the NAT machine dropped large number of packet due to excessive CPU load The overloaded CPU also resulted in huge latency increase and throughput reduction. Figure 5.6 showed throughput with linear increased number of flows from 10k to 100k.

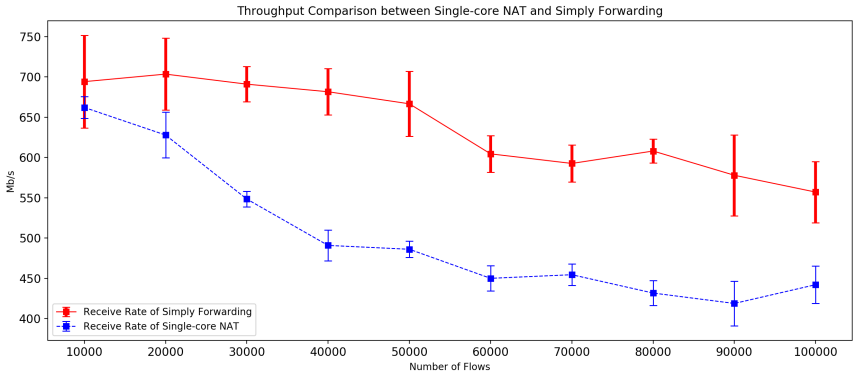

Figure 5.6: Throughput measurement results of single-core NAT and simply forwarding with linearly increased number of flows

This figure gave details of the huge decrease between 10000 flow and 10000 flow, which was a linear descent. The reasons why single-core NAT had such throughput results were explained in section 5.2.3.

5.2.3 Profiling on single-core NAT

The results of latency and throughput shown above were not enough to draw a conclusion. System profiling was needed to evaluate the NAT performance and determine the internal causes of single-core NAT's bottleneck. Since single-core NAT was achieved by click script, no CPU running process but click was valued.

The system profiling could provide detailed information about CPU utilizations of each click function.

There were several steps of the profiling process:

1. Recording the system performance for 50 seconds and remeasuring it for 5-10 times as the single-core NAT script is running.

2. Using perf report to view these results, screening the abnormal ones, and gaining the most desirable data.

3. Using the Flame Graphs to produce the corresponding flame graph. (Figure 5.7, 5.8)

4. Doing the comparison of performance analysis under different scenarios by generating the red/blue differential flame graphs, the meaning of this graph will be explained later. (Figure 5.9)

Profiling graphs of 1 flow and 20000 flows in UDP were used to illustrate what influenced the performance of single-core NAT. Figure 5.7 and 5.8 were the flame graph under 1 flow and flame graph 20000 flows respectively. Both figures only displayed click part of the profiling results.

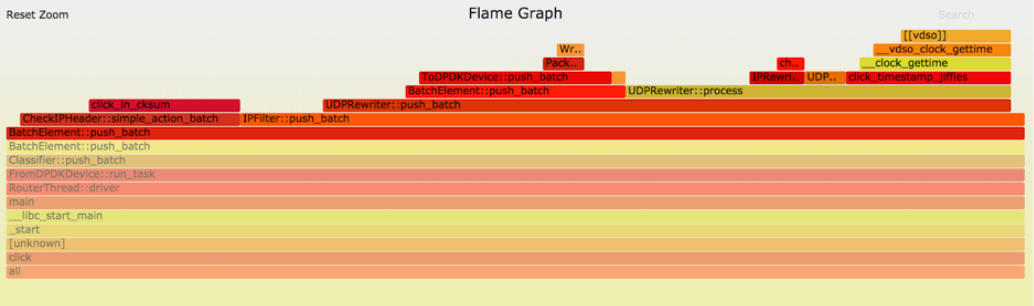

Figure 5.7: Flame Graph of NAT machine stressed with 1 Flow

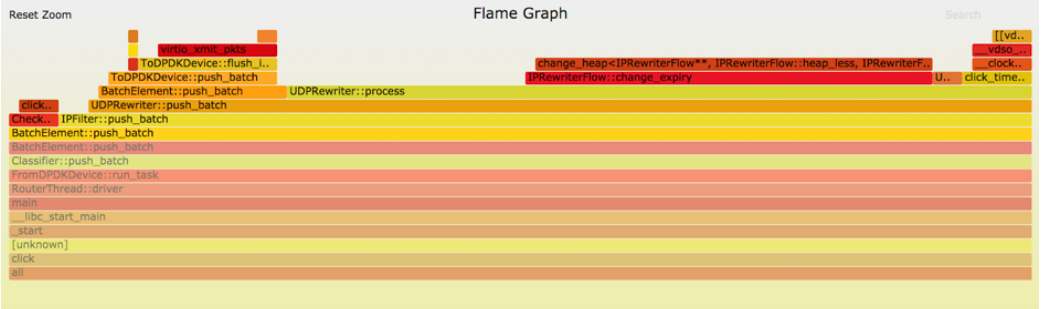

Figure 5.8: Flame Graph of NAT machine stressed with 20k Flow

As those two figures shown above, one Element, UDPRewriter, consumed most of the CUP resource belonging to click functions. Compared to Figure 5.7, there was an increase of CPU consumption on the function named change_heap process calling by the change_expiry function inside IPRewriterflow in Figure 5.8. In 1 flow situation, the system profiling showed that change_heap process only occupied 0.02% of the total CPU resource, while in 20000 flow situation, this ratio has increased to 0.34%.

According to the feature of FlameGraph, the bar of the change_heap process had reached the “plateaus”, which meant it may be the cause of the performance bottleneck. Thus, the conclusion was that the single-core NAT system had the bottleneck because of change_heap process.

However, to deal with the performance regression problem, it was necessary to continuously switch the contrast in different periods and scenes to find out the problem. Introducing the Red/Blue Differential Flame Graphs into the project would be intuitive.

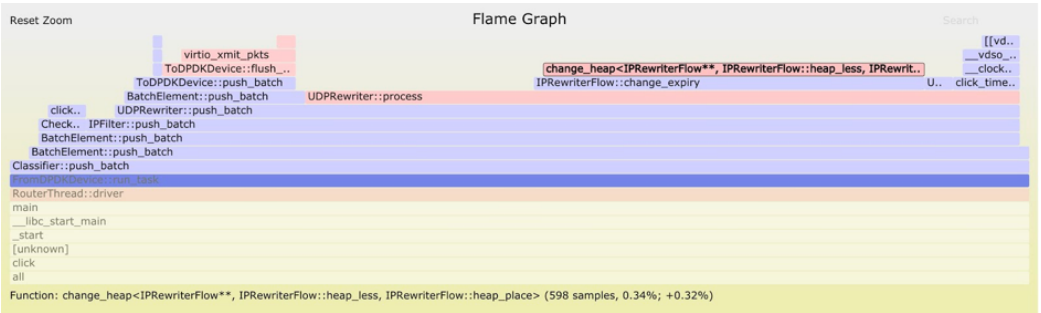

Figure 5.9: Red/Blue Differential Flame Graph of 20k UDP flows

The shape and size of each bar in this flame graph were the same as the CPU flame graph corresponding to the 20k flow data file. The principle of coloring was that if the stack frame has appeared more frequently in a NAT system, it is marked in red indicates the growth, otherwise, it is marked in blue indicates attenuation. As marked in Figure 5.9, the change_heap process has increased by 0.32% which is the main growth in UDPRewriter, this result matches with our conclusion.

The change_heap function increased significantly as the number of flows increased to 20k. In order to figure out how the function changed, the utilization of change_heap function was measured with the increasing number of flows from 1 to 20k. The following figure 5.10 shows the result.

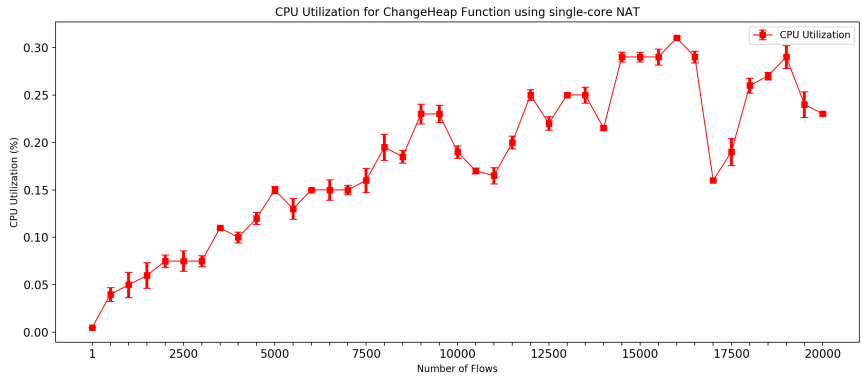

Figure 5.10: CPU Utilization of Change-Heap Function

The utilization result of function increased linearly, but there was some decrease at several points. These points were also the ones that produced the maximum peak in the latency graph.

Above all, the performance of single-core NAT encountered the bottleneck after reaching a certain number of flows. Especially in some specific flow number, the performance declined dramatically. This completely failed to meet ISP's requirements and was not suitable to deal with high concurrency and high traffic network environment. In order to change this, the following sections will give out methods on multi-threaded NAT and the results of their performance.

5.3 Task II Results and Analysis

In this section, we will give out and compare the performance and evaluation metrics of multi-core NAT with previous single-core NAT. And also, we will discuss the difference between each approach we have mentioned above. As the performance of TCP flow is really similar to UDP, in the following measurements, we will mainly focus on UDP traffic flow.

5.3.1 Performance of Global Locked Table

In NAT, spinlocks prevent different CPU threads from manipulating and updating data in Flow tables at the same time, so as to avoid data damage and loss. In other words, after receiving packets, each CPU waits and updates hashtable and heap tree one by one. The following figure 5.11 showed the latency comparison between single-core NAT and 2-core SpinLocked NAT.

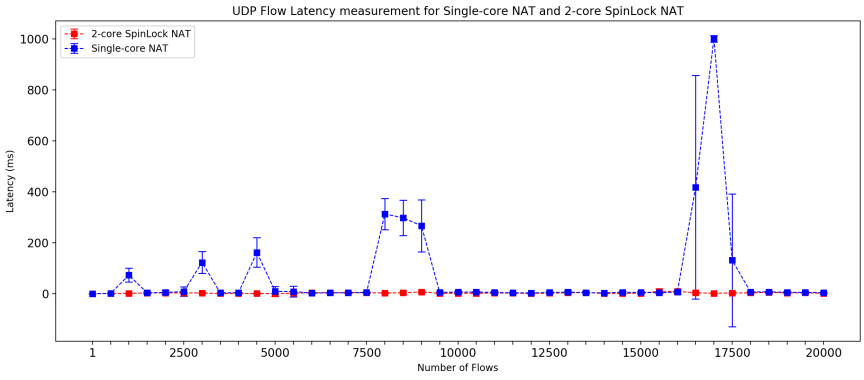

Figure 5.11: Latency Comparison between single-core NAT and 2-core spinlock NAT

From the graph, it was obvious that, when using 2-core Spinlock NAT, the peak maximum value which appeared in the single-core disappeared. Compared with the single-core NAT, the latency of the two-core Spinlock NAT decreased dramatically and the curve was much smoother. Figure 5.12 below showed all the latency value of 2-core SpinLock NAT with different number of flows.

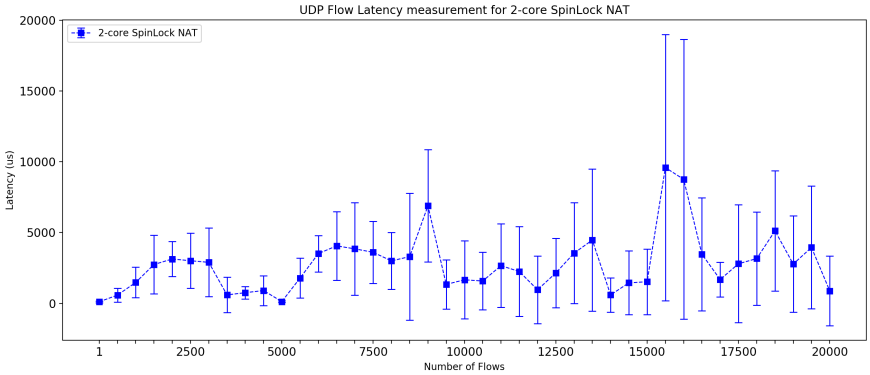

Figure 5.12: Latency Measurement of 2-core spinlock NAT

Latency values at some points in the graph were still very unstable. The maximum latency occurred at point 15500 flow and 16000 flow. But the value was still not as big as the latency of single-core NAT. In general, two-core Spinlock NAT improved a lot compare to the single-core one.

5.3.2 Per-core duplication, Software Classification

The main idea of per-core duplication was to assign each CPU thread a unique rewriter flow table. After receiving packets randomly, all CPU could work and process packets separately. The following figure 5.13 showed the latency comparison between single-core NAT and 2-core NAT with software classification.

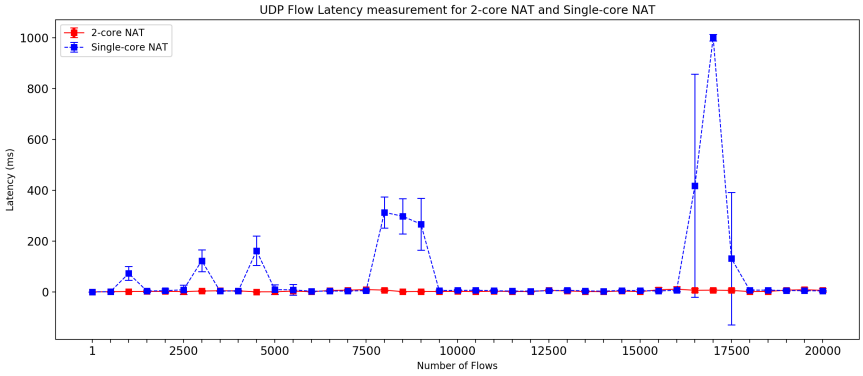

Figure 5.13: Latency Comparison between single-core NAT and 2-core NAT with software classification

This figure was really similar to the one we discussed above, 2-core NAT removed all the peak maximum points and became smoother compared to the single-core NAT. Figure 5.14 below showed all the values of 2-core NAT.

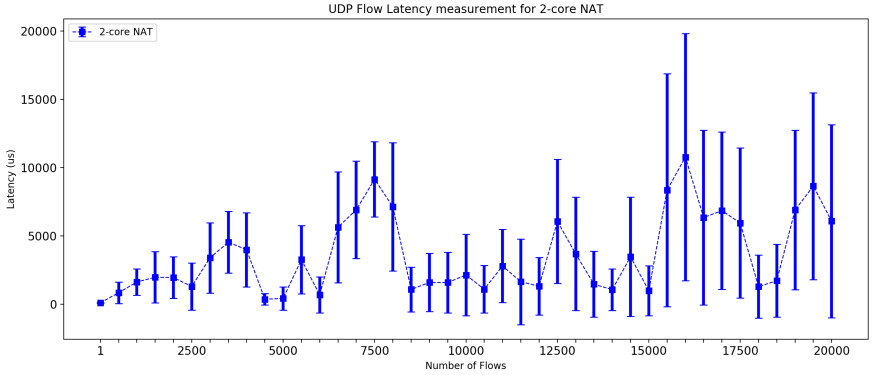

Figure 5.14: Latency Measurement of 2-core NAT with software classification

The latency was still not so stable when the number of flows became larger. But the maximum latency value was also much smaller than that of single-core NAT. The figure showed several small growth at some specific points. These points also coincided with the point where the maximum value of single-core NAT appeared.

5.3.3 Comparison between 2-core and 4-core NAT

Based on previous experiments, it was apparent that, when CPU cores were added to the NAT, the efficiency and performance of NAT were significantly improved. So we went a step further and compare what happens to NATs with more CPU cores. The following figure 5.15 showed latency comparison between 2-core and 4-core NAT with software classification.

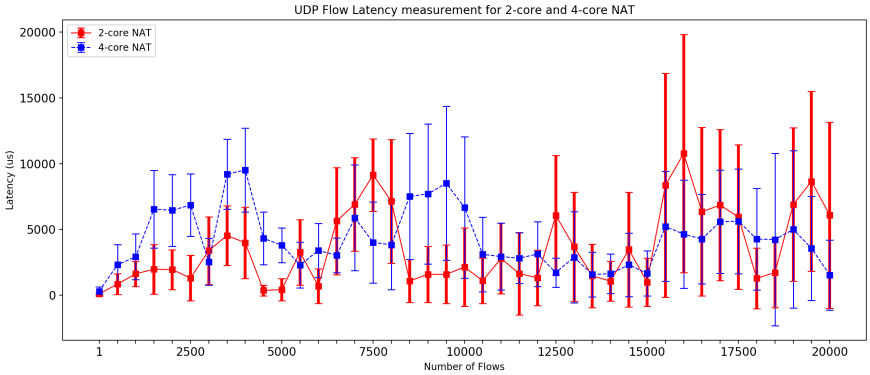

Figure 5.15: Latency Comparison between 2-core NAT and 4-core NAT both with software classification

The values of latency between 2-core and 4-core was really similar. When the number of flows was small, latency data of 2-core was more stable than that of 4-core. However, when the number of flows increased to more than 10k, the situation was exactly the opposite. Also, the following figure 5.16 gave out the comparison between 2-core spinlock NAT and 4-core spinlock NAT.

The result was now slightly different from the previous figure. In general, 2-core Spinlock NAT performed better than the 4-core one. Because of spinlock's working mechanism, each CPU queried and waited for the lock to be acquired. The increase of the CPU core can only the speed of receiving packets, but not greatly improve the efficiency of query and rewriting packets.

In next section, we will compare the difference between 2-core NAT using software classification and Spinlock.

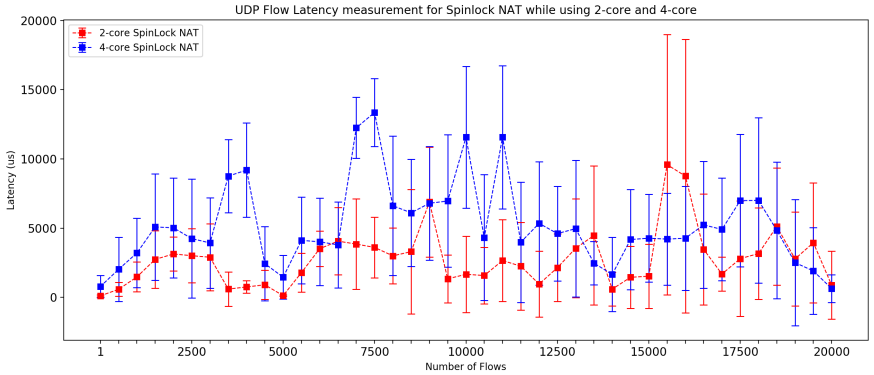

Figure 5.16: Latency Comparison between 2-core spinlock NAT and 4-core spinlock NAT

5.3.4 Comparison between Multi-core Software Classification NAT and SpinLock NAT

As there were two different approaches to realize multithreading, it was interesting to compare the difference of them. The following figure 5.17, 5.18 and 5.19 show the latency and throughput measurement between two methods under 2 CPU cores. The latency measurement of 2-core NAT and 2-core spinlock were almost identical. But Spinlock NAT was slightly better at some points.

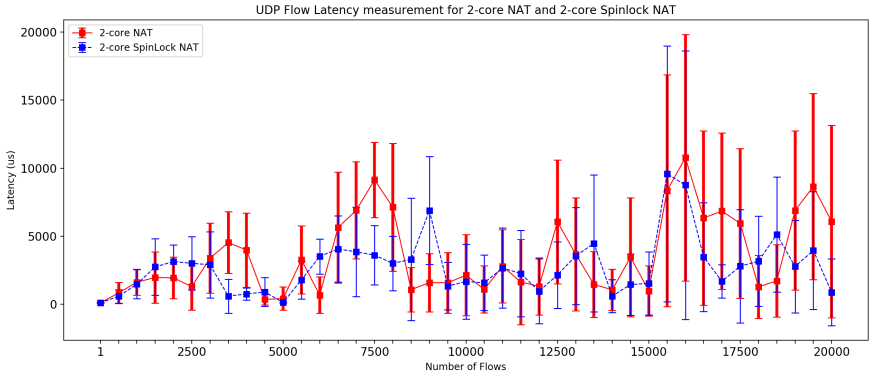

Figure 5.17: Latency comparison between 2-core NAT with software classification and 2-core spinlock NAT

After comparing the throughput plot below, it was clear that spinlock NAT was not as good as 2-core. Throughput of Spinlock was nearly half of the throughput of 2-core NAT. For each thread, while waiting for entering the flow table, some packets were discarded due to the full queue.

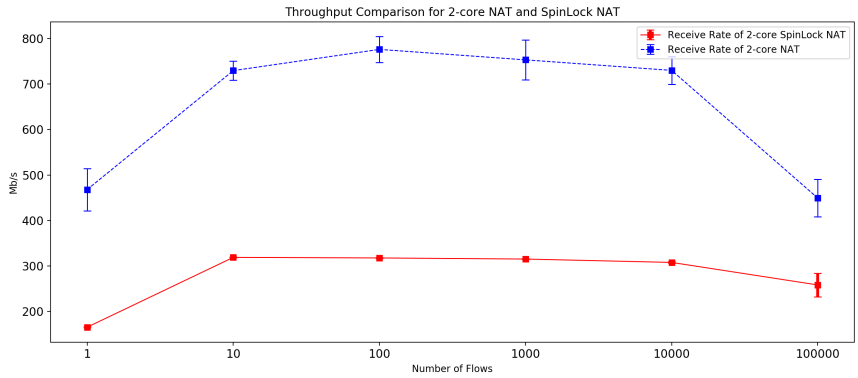

Figure 5.18: Throughput comparison between 2-core NAT with software classification and 2-core spinlock NAT under exponentially increased number of flows

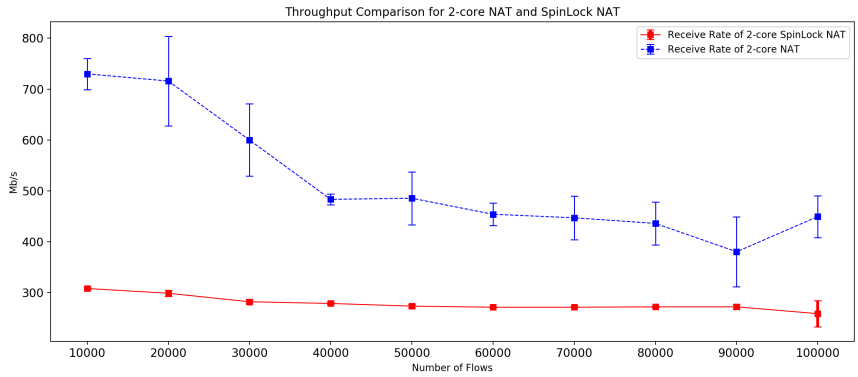

Figure 5.19: Throughput comparison between 2-core NAT with software classification and 2-core spinlock NAT under linearly increased number of flows

Generally speaking, based on the results from the above experiments, software classification performed better in most cases. However, Spinlock NAT was much simpler to implement. If using spinlocked environment, we could directly use

single-core NAT script but assigning different number of CPU cores to realize multi-threading, which software classification method cannot.

5.3.5 Performance of Per-core duplication, with Hardware Classification

The part of hardware classification was really similar to software classification. But the main idea of it was to use hardware capabilities of the NIC to classify packets and direct them towards the right hardware queue. The following figure 5.20 and 5.21 showed the latency and throughput comparison between software classification and hardware classification.

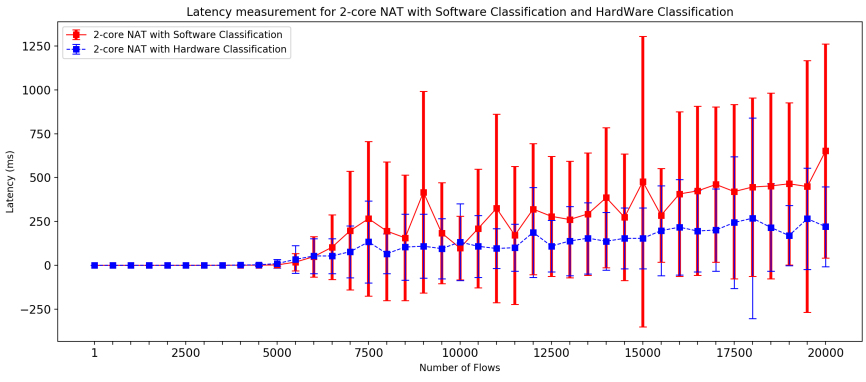

Figure 5.20: Throughput measurement between 2-core NAT with software and hardware classification

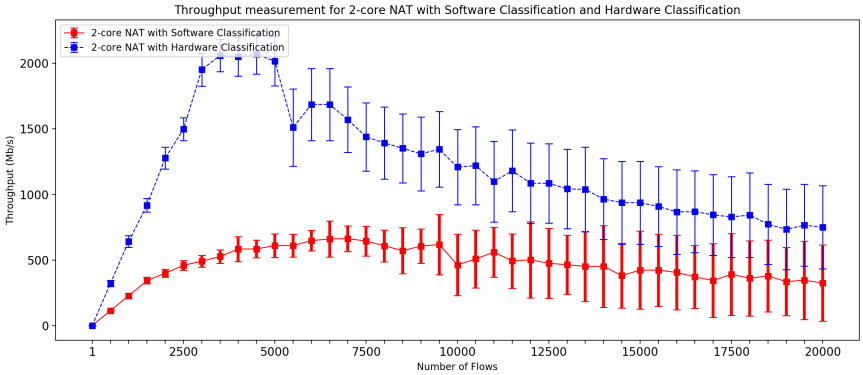

Figure 5.21: Latency comparison between 2-core NAT with software and hardware classification

The performance while using hardware classification was slightly improved. In the best case, NAT throughput reached nearly 2Gb/s with hardware classification and about 500Mb/s when software classification was used only.

Also for the latency, the trend was similar to that of throughput. When the number of flows increased, NAT without hardware classification had a slightly larger delay growth. After assigning cores to receive and classify packets separately, other cores concentrated on address rewriting without considering other things, which improved the performance of NAT.

5.4 Improvements by removing Heap Tree

As described in section 4.3, FastClick uses a heap tree to deal with the time expiry. But, for a multi-threaded NAT, heap tree became a big problem. It is really hard to make multi-threaded safe and only can be maintained by cores separately. In order to deal with this problem, we decided to remove heap tree and introduce a new mechanism to deal with flow timeout. Actually, it can be done directly in the hashtable.

In FastClick, every element stored in the hashtable is added to the head of the link list. After rebalancing, if the link list is over the length threshold, a NAT can delete all the flow elements at the end of the list to ensure the complexity of update and lookup. Also, every flow element queried is changed to the head of the list, so that the active flow will continue to be placed at the front of the list. This will also

speed up update and query speed. The figure 5.22 below showed the result after modifying the code of FastClick.

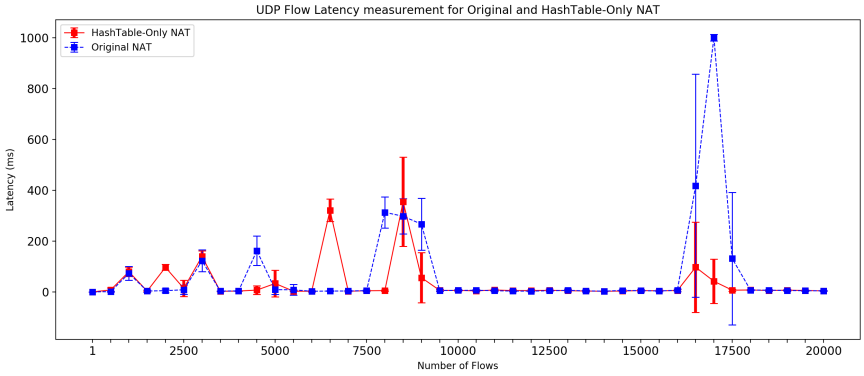

Figure 5.22: Latency comparison between original and HashTable-only NAT

The improvements was really obvious, especially at 17000 flow. But this kind of mechanism was not so efficient while dealing with small number of flows. At 6500 flow point NAT showed an unexpected huge peak.

5.5 Reliability Analysis

The measurements of latency and throughput were repeated by 3 members of the team and each of them collected the sample as section 3.2 stated in different times. Every test was repeated over 100 times to avoid noise and distinguish the data. Besides, data collected from Generator/Receiver was processed in the same way. The plots of 3 different data collections were similar, which proved that the observations were correct. The peaks in latency plots were not caused by artificial errors. Also, the latency and throughput were constant when the NAT machine was replaced by a bridge. It proved that latency and throughput of NAT changed with flow number rather than network condition.

5.6 Validity Analysis

According to the latency plot of single-core NAT, there were multiple peaks. The appearance of these extreme data seemed to follow a specific rule. The profiling part determined the function that caused the extreme latency data and

the cause of the regular peaking pattern are described here. Compared to DPDK-forwarding script, single-core NAT script added an element used for IP address transformation. In this element, single-core NAT checked the hashtable and found the corresponding IP addresses for different flows. The hashtable had 63 buckets at the beginning and changed with the number of flows according to the following equation:

$$if \quad (n_{flow} \geq 2 \times n_{bucket}\prime) : \\ n_{bucket} = 2 \times (n_{bucket}\prime + 1) - 1 \tag{5.1}$$

$n_{bucket}$is the number of buckets, $n_{flow}$is the number of total flows and $n_{bucket}\prime$ is the previous number of buckets. This equation means that the total buckets number changed when the number of flows is more than 2 times of the bucket number. Besides, all flows were rehashed and the heap deleted the time-out flows after bucket number changed which could lead to significant increase in network delay. Figure 5 visually shows the variation between the number of buckets and the number of flows. The single-core NAT latency values were also plotted in the same figure for comparison.

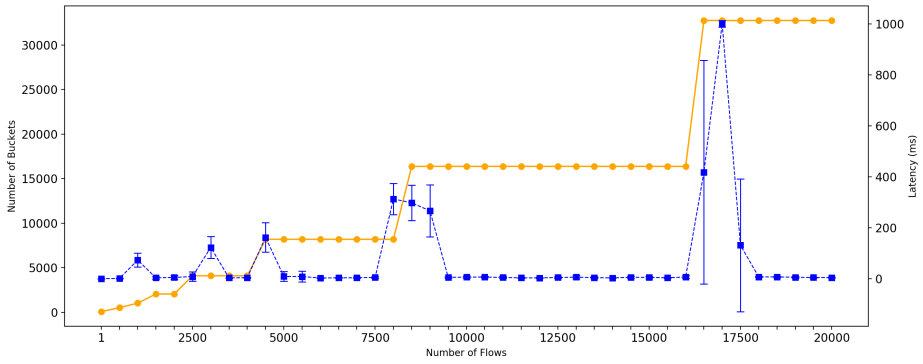

Figure 5.23: Number of buckets inside the HashTable while facing different number of incoming flows compare with latency of single-core NAT

According to the figure, the peaks appeared just around the change points of bucket number. As stated in section 5.3, the latency of multi-threaded NAT had the same peak locations as single-core NAT. For example, single-core NAT met one peak when 17000 flows and 2-core NAT also had a latency peak there. Actually, 2-core NAT stored one hashtable in each core and each hashtable stored 8500 flows when 17000 flows passed through it. As shown in the figure below, the bucket

number changed at 8500 flow. Thus, the rules of bucket number explained the location rule of the latency peaks and the location similarity between single-core NAT and multi-threaded NAT which proved the validity of latency values.

5.7 Discussion

This project focused on discussing different ways to realize NAT multithreading, and tried to further improve the NAT algorithm on the basis of experiments. In this section we will make some comparisons.

For spinlock and software classification methods, there was little difference in latency measurement. However, in throughput, spinlock lost a lot of packets at some time, which reduced the receiving rate to half of the normal value. This was most likely due to the CPU's busy-waiting for the preamble CPU to release the lock when it needed to access the flow table. In addition, when queuing for updating a single flow table, the waiting time of each CPU may also showed a straight upward trend. There were many ways to solve this problem. For example, we could only lock every bucket. When the CPU tries to query and update the mapping information of incoming flow, the kernel only locks the corresponding bucket in the table. This could further refine spinlock, but it would also make the code more complex.

Software classification performed better generally. However, because we needed to create a corresponding number of flow tables in memory, when the stream storage was not balanced enough, it caused a certain amount of memory loss. In addition, since we redefined rewriter elements for each CPU, the Click code for configuring this method were more cumbersome than spinlock.

When we used Hardware Classification, because of the introduction of ACK and stateful connection, it was no longer efficient to use only two-core software classification to complete both receiving and updating work. Relatively, because Hardware classification directly used a separate CPU to do the heavy packet allocation work, it eased the workload of processing the package and made the performance greatly improved.

In terms of the number of cores, there was a tremendous performance improvement from single-core to two-core. However, when the number of CPU cores is further increased, the processing performance of NAT didn't necessarily continue to improve. Due to the impact of task conversion efficiency of CPU collaborative processing, the transfer of packets between different CPUs may have resulted in a

certain loss of efficiency.

After studying the causes of NAT bottleneck, we tried to further improve FastClick source code. Then a new method was used to deal with timeout. Although it had some advantages, this method needed further research and adjustments. It can be the direction of future improvement of this project.

Chapter 6

Conclusions and Future Work

6.1 Conclusion

As the report illustrated, the performance of multi-threaded NAT is significantly improved compared with that of single-core NAT, especially in the latency measurement. Multi-core NAT can decrease the latency from about 1s to only 10ms in some specific number of flows.

Besides, this report discusses different strategies of implementing multi-threaded NAT and gives out the performance comparison among these methods. Based on the results of our experiments, deploying spinlock in multi-threaded NAT improves the latency improvements as much as per-core duplication methods. But, compared to other multi-threaded strategies, spinlock leads to worse throughput performance. There are two per-core duplication strategies to implement multi-threaded NAT. Software classification performs fairly in latency improvement and has better throughput results than spinlock. Hardware classification strategy has more stable latency results and much higher throughput than simply software.

In addition, based on the limitations of NAT, we have made some improvements to the source code of NAT. By removing Heap tree and adding a new timeout mechanism, the extreme latency values disappear.

In order to make further enhancement, various hashtable algorithms in NAT have been searched and studied. We also summarized them on this paper to allow for further improvements.

6.2 Limitations

- The first and biggest limitation comes from the testbed. There are multiple available packets generators but few of them could meet our demands. To pressure the NAT, we need that generator could send out millions of flows with high speed. However, those packets generators couldn't support either high-speed mode or multiple flows when they run on the VM. If the testbed was based on physical machine, we would get more accurate results and approach the bottleneck of single-core NAT.

- Multi-core co-operation reduced the effectiveness of each core, compared to single-core NAT. It may be caused by FastClick or virtual machines.

- Another limitation comes from our limited knowledge of C++ and FastClick. The third task required us to finish the best method that we have found. The method we chose was achieved in another software named VPP but we needed to find the algorithm within thousands of coding files. Besides, to implement this method in FastClick, we ought to be familiar with the FastClick framework. But, due to time and knowledge limitation, we weren't able to implement the best method.

6.3 Future work

Obviously, based on our results, this project can go further in several ways:

- Source code: Because the traditional hashtable and heap tree processing performance is still inadequate, the NAT implementation code in FastClick can be improved in the future. For example, using other hashtables, such as Bihash, Cuckoo hash, or other collation and induction methods which are more effective in dealing with the problem of flow time expiration. All of these can greatly improve NAT and FastClick modules.

- Hardware Generator: Although generators we discussed in Section 2.3 have limited performance on VMs, they can generate huge amount of data in a short period to test NATs. Deploying hardware generators can make tdata collection much easier and more reliable.

- Thread-safe algorithm: There are more strategies to build a multi-threaded NAT. It's meaningful to find more algorithms and compare them with our achieved ones.

6.4 Reflections

Our work, including the literature and the experiments, didn't involve any public information. All related works had been cited appropriately in the reference part and all pictures were our original work. The experiments required no biological and chemical agents which may lead to environment issues.

However, the server where our test virtual machines run may requires much energy for our experiments because the softwares, both FastClick and T-Rex, need to consume much network resource to generate thousands of high-speed flows. The electricity consumption of our experiments is prodigious. Great electricity consumption may lead to the concerns about global warming issue, which is one important cause of the catastrophic climate changes. But compared to others' experiments running on the hardware testbed, our experiment was done on testbed built on virtual machine and required electricity as its only resource, which led to much less material waste like electronic components.

Bibliography

[1] K. B. Egevang and P. Francis, "The ip network address translator (nat)," Internet Requests for Comments, RFC Editor, RFC 1631, May 1994, http://www.rfc-editor.org/rfc/rfc1631.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1631.txt

[2] P. Srisuresh and M. Holdrege, "Ip network address translator (nat) terminology and considerations," Internet Requests for Comments, RFC Editor, RFC 2663, August 1999, http://www.rfc-editor.org/rfc/rfc2663.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2663.txt

[3] J. Postel, "Internet protocol," *Internet Request for Comments*, vol. RFC 791 (Standard), Sep. 1981, updated by RFC 1349. [Online]. Available: http://www.rfc-editor.org/rfc/rfc791.txt

[4] K.-J. Beasley, "Design and implementation of a network address translator," 2004.

[5] S. Perreault, I. Yamagata, S. Miyakawa, A. Nakagawa, and H. Ashida, "Common requirements for carrier-grade nats (cgns)," Internet Requests for Comments, RFC Editor, BCP 127, April 2013, http://www.rfc-editor.org/rfc/rfc6888.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6888.txt

[6] D. Intel, "Data plane development kit," 2014.

[7] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *Architectures for Networking and Communications Systems (ANCS), 2015 ACM/IEEE Symposium on*. IEEE, 2015, pp. 5–16.

[8] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000.

[9] K. Wiles, "The pktgen application," 2015.

[10] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *Proceedings of the 2015 Internet Measurement Conference*. ACM, 2015, pp. 275–287.

[11] P. Srivats, "Ostinato packet/traffic generator and analyzer," 2011.

[12] R. Jones *et al.*, "Netperf: a network performance benchmark," *Information Networks Division, Hewlett-Packard Company*, 1996.

[13] Cisco, "T-rex introduction manual." [Online]. Available: http://trex-tgn.cisco.com/trex/doc/trex_manual.html

[14] "Linux kernel profiling with perf-tutorial." [Online]. Available: https://perf.wiki.kernel.org/index.php/Tutorial#Introduction

[15] "Flame graphs." [Online]. Available: http://www.brendangregg.com/flamegraphs.html

[16] "IEEE code of ethics. (n.d.)." [Online]. Available: https://www.ieee.org/about/corporate/governance/p7-8.html

[17] P. Gupta and N. McKeown, "Algorithms for packet classification," *IEEE Network*, vol. 15, no. 2, pp. 24–32, 2001.

[18] "Vector packet processing." [Online]. Available: https://docs.fd.io/vpp/19.01/

[19] V. Srinivasan, "A packet classification and filter management system," in *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3. IEEE, 2001, pp. 1464–1473.

[20] P. Gupta and N. McKeown, "Packet classification on multiple fields," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 4, pp. 147–160, 1999.

[21] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, *Fast and scalable layer four switching*. ACM, 1998, vol. 28, no. 4.

[22] Z. Cao, Z. Wang, E. Zegura *et al.*, "Performance of hashing-based schemes for internet load balancing," in *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies*. IEEE, 2000, pp. 332–341.

[23] J. Van Lunteren and T. Engbersen, "Fast and scalable packet classification," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 4, pp. 560–571, 2003.

[24] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 4. ACM, 1999, pp. 135–146.

[25] P.-C. Wang, C.-T. Chan, S.-C. Hu, C.-L. Lee, and W.-C. Tseng, "High-speed packet classification for differentiated services in next-generation networks," *IEEE Transactions on Multimedia*, vol. 6, no. 6, pp. 925–935, 2004.

Appendix A

NAT Algorithm Study

Algorithm	Description	Performance	Fast Updates	Paper/Resource
Hardware-based solution (TCAM)	It is a triple content access memory which can realize fast and high-volume data for parallel search.	Has the time complexity of $O(1)$. Fast but required a large storage CAM.	Yes	A packet classification and filter management system [19]
Set-pruning tries	Can be used in the occasion that the dimension is no more than 2 and rules are complex.	$O(dW)$ where d is the dimension of rules	No	Packet classification on multiple fields[20]
Grid-of-tries	The search process is divided into two steps. The first step is to match the dest IP with the longest match prefix, and then find the least expensive filtering principle by looking up the corresponding src IP trie tree.	$O(W^{d-1})$ This algorithm supports 2-dimensional classification rules only and has large memory consumption, which leads to high difficulty in the update stage.	No	Fast and scalable layer four switching[21]

continued on next page

continued from previous page

Algorithm	Description	Performance	Fast Updates	Paper/Resource
Cross-Producting	It divides the search space according to the rule segmentations along each dimension. With high throughput, but memory requirements are too large.	The search can be done quickly in a parallel lookup. The time complexity is $O(dW)$ where d is the dimension of rules	No	Fast and scalable layer four switching[21]
Recursive Flow Classification	This algorithm is fast in classification, its storage space is related to dimensions and has large memory consumption. Required parallel computing.	Convert one mapping to a multi-phase mapping, each mapping called a reduction. The time complexity related to the dimensionality of the search space $O(P)$.	No	Performance of hashing-based schemes for internet load balancing[22]
Parallel Packet Classification	This algorithm searches all the rules set in parallel. so the time is decided by the maximum field-search time.	$O(\frac{W}{s}+1)$ where W' is the largest field size, $s(stride) = 8$ bits	yes	Fast and Scalable Packet Classification[23]

continued on next page

continued from previous page				
Algorithm	Description	Performance	Fast Updates	Paper/Resource
Tuple Space (Rectangular search)	A tuple-space-based algorithm, highly scalable with markers and pre-computation to eliminate subsets of tuple space after each probe.	Has memory-explosion problem, $O(2W-1)$ lookup time complexity, where W is the length of IP address	Yes	Packet Classification Using Tuple Space Search[24]
Tuple Reduction (Space optimization & Speed optimization)	Improved from rectangular search by reducing the number of tuples. It's a combination of the semi-optimization algorithm and sparse-region reduction.	The paper didn't propose an actual time complexity on this algorithm, but from the table, we can conclude the time is reduced from $\frac{1}{3}$ to $\frac{1}{2}$ of the rectangle search.	Yes	High-Speed Packet Classification for Differentiated Services in Next-Generation Networks[25]

Table A.1: Summary of Packet Classification Algorithm

Schemes	Algorithms	Description	Lookup Time	Update Time
Linear Search	Linear Search	This method is the basic data structure of all, it keeps the prefixes in a linked list, and traverse one at a time.	$O(N^*)$	$O(1)$
Binary Search	Binary Search on prefix lengths	The algorithm first probes the hashtable of the half length prefix. If a node is found in this hashtable, there is no need to probe tables of the prefix before. If no node is found, hashtables need not be probed the prefix behind. The remaining hashtables are similarly probed in a binary search manner.	$O(logN)$	$O(log(\frac{N}{K}))$ on the levels of a 2^k-ary trie
Binary Search	Binary Search on intervals represented by prefixes	This method is finding the narrowest enclosing interval of the point represented by the address. In this case, the interval means a contiguous range , longer prefixes will represent shorter intervals.	$O(log2N)$	-
Hashing	Hash search	The hash lookup algorithm is based on mask traversal to achieve strict longest prefix matching (at least 32 times to get the result)	Requires O(W) hash lookups, independent of the number of prefixes.	Range from different method

continued on next page

continued from previous page

Schemes	Algorithms	Description	Lookup Time	Update Time
Tries	Binary Trie	The basic idea of these algorithms is to construct a binary tree based on the binary bits of the prefix value. When searching, use the target address as an index and traverse in the binary tree.	$O(W)$	$O(W)$
Tries	Patricia	PATRICIA tries are radix tries with radix equals 2, which means that each bit of the key is compared individually and each node is a two-way branch.	$O(W^2)$	$O(W)$
Tries	Lulea Trie	The Lulea Trie algorithm is using large strides in fixed stride multibit tries will result in a greater number of contiguous nodes with the same best matching prefix and next-hop information. The key idea of the Lulea scheme is to replace all consecutive elements in a trie node that have the same value with a single value, and use a bitmap to indicate which elements have been omitted.	$O(\frac{W}{k^{\dagger}})$	Need reconstruct

continued on next page

continued from previous page				
Schemes	Algorithms	Description	Lookup Time	Update Time
Tries	Level Compressed tries (LC-Tries)	It is suitable for a large routing table, which is quite fast compared to the original hash implementation. The price is the increase of internal consumption and complex noise	$O(W/k)$	Not supported
Tries	8-8-8-8 mtrie tree	Compared to hash, the complexity of 256-way mtrie is fixed (since it not relevant to the number of items in the table) and smaller.	The time complexity of mtrie tree defined as $O(\frac{W}{8})$	Update complexity is $O(\frac{W}{k+2k})$

Table A.2: Summary of Routing Lookup Algorithm

* k is the stride value in multi-ary trie, W is the length of IP address † N is routing table item number