Comparison of flow classification methods for Load Balancing

IK2200 - Communication System Design
Final report of group project

Students
Davide Brunello
Devinoraya Radityapalma
Elisabeth Mintchev
Marc Viñas
Massimo Girondi
Omar Giordano

Course responsible and examiner
Dejan Manojlo Kostic
Teaching Assistant
Tom Barbette
Academic supervisor
Håkan Olsson

Royal Institute of Technology
School of Electrical Engineering and Computer Science

Stockholm - January 7, 2020

Abstract

Load balancing is a technique used in computer and communication systems that consists of distributing the processing load of a specific service between multiple servers. This technique is particularly appropriate to improve throughput and latency performance for web services, where multiple servers respond to the requests of millions of users.

Load balancing is performed through a device, hardware or software, called Load Balancer which is located between the outside world and the server pool. Its purpose is to accept all the destined requests for the unique IP address of the service and redirect them to one of the servers.

In this work, we implemented and investigated a software Load Balancer based on FastClick and DPDK. Load Balancers with different stateful methods have been implemented in a multi-core environment and compared with a stateless implementation.

The results showed that using a different flow table per-core enables reaching the highest throughput in a stateful scenario.

Furthermore, a state-of-the-art method for managing flow tables has been studied and implemented. The first results obtained in terms of throughput and RTT seem comparable to the ones obtained with the methods implemented in DPDK. However, a further analysis is required to obtain scientifically valid results.

Finally, a digression about Hardware-Based flow classification has also been performed, even if no clear results have been achieved.

Keywords

Load Balancer, Click, FastClick, multi-core, stateful LB, stateless LB, Internet technologies, packet-classification, high-speed networking, DPDK

Acknowledgements

We would like to thank professor Dejan Manojlo Kostic for having allowed us to work on this project.

We are thankful to professor Håkan Olsson for the supervision and help that he provided us with regards to the writing of this document. We are grateful to Tom Barbette, our TA, for his constant support and help to overcome the problems regarding the implementation and the testbed.

Finally, we would like to thank SFI and NVM groups for the weekly peer review we received as it helped us to improve our final work.

Stockholm, January 7, 2020

Contents

Abstract i

Acknowledgements iii

List of Figures viii

1 Introduction 1

1.1 Problem . . . 1

1.2 Purpose . . . 2

1.3 Goals . . . 2

1.4 Research Methodology . . . 3

1.5 Delimitations . . . 4

1.6 Structure of the report . . . 4

2 Background 5

2.1 Load Balancers . . . 5

2.1.1 Stateless Load Balancer . . . 6

2.1.2 Stateful Load Balancer . . . 6

2.2 Data Plane Development Kit . . . 7

2.3 Non-Uniform Memory Access . . . 7

2.4 Click and FastClick 7
2.5 Hashing and collision in hash table 8
2.6 Related work 8
2.6.1 Binary Tree based flow classification methods 8
2.6.2 Hash based flow classification method 9
2.6.3 Summary 10

3 Measurement technique **11**
3.1 Research Process 11
3.2 Research Paradigm 12
3.3 Experimental design 12
3.3.1 Testbed 12
3.3.2 Hardware and Software used 12
3.4 Data Collection 13
3.4.1 iPerf3 and the first round of tests 14
3.4.2 TRex 14
3.4.3 TRex result analysis 15
3.4.4 Hacking TRex 16
3.5 Profiling 17
3.6 Assessing the reliability and validity of the data collected 17
3.7 Planned Data Analysis 18
3.7.1 Data Analysis Technique 18

4 Setup and Implementation **19**
4.1 Testbed setup 19

4.2 Implementation of the methods 20
4.2.1 Elements description 20
4.2.1.1 Common elements 21
4.2.1.2 Router only elements 22
4.2.1.3 LB only elements 22
4.2.2 Global locked flow table 23
4.2.3 DPDK's Cuckoo hash table 25
4.2.4 Per Core Duplication 26
4.2.5 Hardware-based classification 27
4.2.5.1 Hardware-only load balancer 28
4.2.6 Cuckoo++ 28

5 Results **30**
5.1 Testbed performances 30
5.1.1 System bottleneck 30
5.1.2 Simple Forwarder Performance 31
5.1.3 Latency evaluation 33
5.2 Major results 34
5.2.1 Stateless LB performance 34
5.2.2 Stateful LB performance 36
5.2.3 Stateful LB using global locked flow table 37
5.2.4 Stateful LB using DPDK RTE_HASH_EXTRA_FLAGS 40
5.2.4.1 `MULTI_WRITER_ADD` 40
5.2.4.2 `RW_CONCURRENCY` 42
5.2.4.3 `MULTI_WRITER_ADD and RW_CONCURRENCY` 42

5.2.4.4 RW_CONCURRENCY_LF 45
5.2.4.5 MULTI_WRITER_ADD and RW_CONCURRENCY_LF 46
5.2.4.6 Comparison and consideration 46
5.2.5 Stateful LB using Per Core Duplication 48
5.3 Reliability Analysis 48
5.4 Validity Analysis 50
5.5 Discussion 50
5.6 Hardware classification and Cuckoo++ results 57
5.6.1 Hardware classification 57
5.6.2 Cuckoo++ 57

6 Conclusions and Future work 58

6.1 Conclusions 58
6.2 Limitations 59
6.3 Future Work 59
6.3.1 Hardware classification implementation 59
6.3.2 TRex measurement mechanisms 60
6.3.3 Flow ageing implementation 60
6.4 Reflections 61

Bibliography 62

Glossary 65

Acronyms 66

List of Figures

1.1 Learning and development approach . . . 3

2.1 Role of the Load Balancer . . . 6

3.1 Testbed configuration . . . 13

4.1 Click router configuration [6] . . . 24

4.2 Stateful LB simplified configuration . . . 25

5.1 Throughput obtained with different number of cores and multiple frequencies configuration . . . 31

5.2 Throughput and Latency of the forwarder with different setups . . . 32

5.3 Throughput and RTT of the simple forwarder with different numbers of cores . . . 33

5.4 Throughput and RTT of the Stateless LB . . . 34

5.5 Profiling of the Stateless LB . . . 35

5.6 Cumulative Distribution Function of the RTT for the Stateless LB . . . 36

5.7 Throughput and RTT of the stateful LB with different numbers of cores . . . 37

5.8 Profiling of the *Stateful* LB . . . 38

5.9 Cumulative Distribution Function of the RTT for the *Stateful* LB . . . 38

5.10 Throughput and RTT of the *Spinlock* LB . . . 39

5.11 Profiling of the *Spinlock* LB 39

5.12 Cumulative Distribution Function of the RTT for the *Spinlock* LB 40

5.13 Throughput and RTT of Stateful with *MW* flag 41

5.14 Comparison between *Stateful* (`Stf NF`, no flag set) and Stateful with `MW` flag 41

5.15 Profiling of Stateful LB with `MW` flag set 43

5.16 Throughput and RTT of Stateful with `RWC` flag 44

5.17 Throughput and RTT of Stateful with `StF MW RWC` 44

5.18 Comparison between `StF MW RWC` and `StF RWC` 45

5.19 Throughput and RTT using `LF` flag 45

5.20 Comparison of Throughput and RTT of *StF MW LF* and *StF LF* 46

5.21 Throughput and RTT comparison between different implementations 47

5.22 Throughput and RTT of the Per Core Duplication 48

5.23 Profiling of Stateful using *Per Core Duplication* method 49

5.24 Cumulative Distribution Function of the RTT for the *Per Core Duplication* method 49

5.25 Throughput and RTT comparison between Stateful methods 51

5.26 Throughput and RTT comparison running with one core 53

5.27 Throughput and RTT comparison running with three cores 54

5.28 Throughput and RTT comparison running with six cores 55

5.29 Throughput and RTT comparison running with eight cores 56

6.1 Throughput over time of the same test ran 3 times 60

6.2 Profiling of LB with Per Core Duplication 61

Chapter 1

Introduction

With the increasing size of Internet traffic, websites and applications need to serve millions of simultaneous requests, return the correct data and meet expectations on fast services. To keep up with these requirements, multiple servers are needed. In order to properly divide traffic among all the servers [1], Load Balancers (LBs) are used. They distribute the load to the available servers and improve the overall performance [2], [3]. With this specific structure, LBs are critical components in the network [2]. In fact, since all the traffic going to a datacenter pass through the LB, this can become the bottleneck of the system, worsening the overall performance.

Typically, LBs are expensive physical machines [4] that have to manage transmitting rates higher than 100 Gbps for one single IP address [3] and need to avoid adding high delays to the transmitted packets. This approach has shown many limitations but the ones which concern to this work are mainly two [2]. First, hardware LBs are very difficult to upgrade, not allowing future improvements for the implementation of new and better performing load balancing methods. Secondly, physical load balancing devices are expensive to upgrade and have a slow adaptability to the network demands. The only way to increase their performance is to deploy more physical LBs, adding additional costs and disabling a potential fast adaptability of the system. In order to overcome these limitations, software LBs[1] have been developed.

Two types of load balancing algorithms exist: stateless algorithms, also known as static, and stateful algorithms also referred to as dynamic [1].

1.1 Problem

Load balancers require particular algorithms to sort the traffic: each client-server flow needs to be managed — and known — by the LB to correctly distribute the traffic to the servers. Current

[1] After the introduction, this document will only take into consideration software LBs.

techniques are based on stateless and stateful models. As it is discussed in detail in 2.1.1, stateless LBs use hash indexing on the source to direct the connection to a specific server, but no state is saved. Since stateless LBs use hashing algorithms to keep the micro-flow (the client-server relation), they do not balance the system load effectively because the servers are randomly chosen through the hashes.

Stateful LBs (discussed in 2.1.2) are the ones that, potentially, could achieve higher balancing performance since they have a better knowledge about the traffic that passes through them. However, this higher precision comes at the cost of a more important computational load requirement, which involves lower throughput and latency performance.

The critical factor with LBs is that they need to route the traffic to the servers as fast as possible. Thus, in order to avoid LB from becoming the bottleneck of the network, the technical characteristics of the LB needs to be higher than the slowest device in the processing chain. In our work, we have analysed and improved the performances of the software. However, it is fundamental to design the hardware architecture in the most optimized way, in order not to lose too much computing power or too much time while processing the data.

1.2 Purpose

The purpose of this work is to investigate different approaches to implement LBs in software with the help of FastClick [5] as a prototyping platform. By using this software it is possible to scale up with multi-core implementation, and to investigate the latest capabilities of high-speed NIC (Network Interface Controller) for in-hardware flow classification. Based on that investigation, we intend to provide suggestions about which method to choose for a certain use. Another aim of our work is the review of state-of-the-art methods to update flow tables and implement the one which seems to be the best and the most relevant. Our work will be beneficial for the development of better performing LBs in the future.

1.3 Goals

Our main goal is to understand, implement and analyse different LB methods, and finally to compare them to understand which ones are better in various contexts and why. This information will be useful to develop more efficient LBs.

This work can be divided into three high-level tasks, with several minor goals. The first objective to be achieved was the correct configuration of the testbed for the experiments. This configuration allowed the implementation of a single-core LB in FastClick. The measuring of throughput and latency, together with the LB profiling, completed the first goal: the study of a single core LB.

The second main goal consisted in the implementation of multiple multi-core stateful load balancing methods. First, by using a global lock around the flow table, we implemented a basic thread-safe mechanism. Then, the same functionality has been implemented through the use of thread-safe data-structures and different algorithms. At last, a hardware-based implementation has been tested in order to exploit the advanced features available in the network cards used.

Finally, to obtain meaningful results, the methods mentioned above have been tested and compared to each other, saving and analysing the obtained results. The last goal has been the analysis of some methods presented in the literature, assessing their pros and cons. Then, one of these methods has been implemented, allowing us to compare it with the others.

1.4 Research Methodology

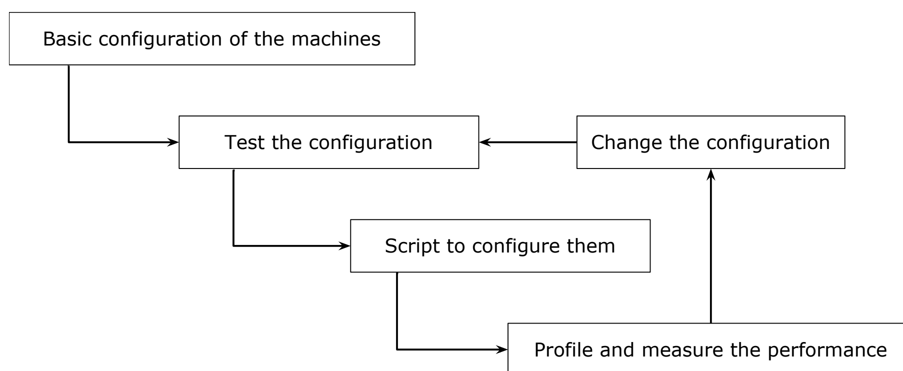


Figure 1.1: Learning and development approach

Our approach consisted of five steps as reported in Figure 1.1, we used empirical methods to obtain quantitative results. After the initial testbed setup, we started with a simple router implementation in Click to become familiar with the environment. This implementation has been derived from the one described in [6]. Next, we implemented the basic configuration for the stateful and stateless LB. Then we used a closed/feedback loop approach to implement, automatize, evaluate and improve the LB. In this way, we have been able to understand which method performed better than the others. Moreover, we created scripts to automatically run the tests and generate the plots of data such as throughput and latency. Our final goal was also to create easily replicable experiments so that other people simply reusing our code and implementation could obtain the same result we achieved. To guarantee this, we used Git as a version control system for the code and we also created and documented scripts to automatically recreate the same environment (testbed) we used in the tests we conducted. By using this approach, we can state that the internal validity of our experiments is high because we had control on most of the background variable of our system (as the number of cores, the frequency, the traffic generated by the system and the bypass of Linux kernel).

1.5 Delimitations

This work intends to compare different methods of stateful load balancing, assess the performance in a multi-core environment, and lastly implement a method to manage the flow table. However, we neither investigated different software LB implementation nor evaluated performances on different hardware. Furthermore, we did not assess the performance of FastClick as NAT or router and we also limited our study to make the LB run only with the DPDK library. Finally, we restricted the literature review of the flow table implementation methods to a few articles, implementing only the one that seemed to be the best to us.

1.6 Structure of the report

After having given an overall introduction to the project, we are now going to explain the structure of the other chapters and, as a consequence, of the entire project.

Chapter 2 provides all the relevant background information to better comprehend the topic, such as the difference between load balancing methods, what is DPDK [7] and why we used it, together with a slightly more detailed explanation about what NUMA [8] is. In the end, we explained several different works related to this topic, giving reasons on why we chose to implement one method with respect to the others.

Chapter 3 presents the research methods used, explaining our methodology. It also includes the testbed description and the explanation of the methods adopted for data collection. In particular, it is possible to find the tools that we used for the analysis of the data we have previously collected.

Chapter 4 regards the implementation part. All the different methods that we used and implemented are explained under a detailed and technical point of view.

Chapter 5 is where all the results have been reported and discussed. This is the section where it is possible to find the graphs about our work and the comparison between the different methods.

Finally, in Chapter 6 we reported the conclusion, reflections about the project and possible future works together with investigations which might be pursued.

Chapter 2

Background

To handle millions of simultaneous network requests, many organizations started using a mechanism called "load balancing" [1, 2, 9].

Load Balancer is an ad-hoc middlebox with the task of "sorting" and distributing the various requests among a set of servers, trying to balance the load evenly. The sorting is usually performed by dedicated and very expensive machines [2, 4] but, many are trying to implement it in software to increase the performance. Hence, common servers can be used and cloud platforms can be exploited to support and deploy these systems [3].

The main critical task performed by a LB is the classification of the flows, which determines how a certain traffic is distributed in the server set. This classification can be divided, mostly, in two categories: stateless and stateful, also known as static and dynamic [10]. Extending the concept, these are the two types of LBs that are most commonly used.

2.1 Load Balancers

LBs are network devices that distribute the traffic load of a specific service between more servers [11]. Their task is to spread the incoming traffic among different servers, solving the problem of associating the clients with the right server. The concept behind the usage of a LB is resource usage optimization, which entails managing network packets as efficiently as possible. LBs set the rules on how a client is served by one server rather than another one (see section 2.1.1 and 2.1.2). By doing so, it is possible to highly increase the overall scalability, the amount of processed traffic, the reliability of the system, and the resource overload avoidance of the entire architecture. As previously mentioned, there are two different types of load balancing techniques: stateless and stateful.

2.1.1 Stateless Load Balancer

Stateless or static LBs ensure the knowledge about which machine a particular client is talking to. Stateless LBs always send the request of a particular client to the same machine in a predictable way. By looking at Figure 2.1, it is possible to better understand the concept: Client A has a request and sends it to the LB. At this point, the LB picks a server using hash principle: basically, it takes the 5-tuple(IPs and ports of source and destination and protocol used) of the arriving packet to select one of the servers. In this way, all the packets from a specific client are always sent to the same server.

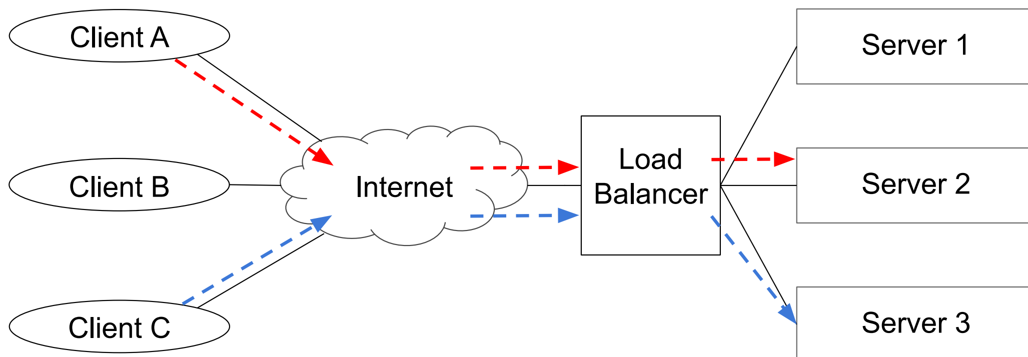


Figure 2.1: Role of the Load Balancer

However, stateless LBs do not consider the real amount of load that each server has. For example, we can look at two servers in a pool, where the first one is overloaded and the second one is free. Now, when a new client sends a request that arrives at the LB, the latter have to compute the hash of the packets. At this point, if the hash corresponds to the first server (the overloaded), the packet is transferred to that server, even if there is a server completely free of load. Therefore, we can state that this is not a good method to perform balancing among the servers.

Another drawback is that the system is not scalable: when the number of servers changes, the hashing that has been previously done is not valid anymore, breaking active connections.

2.1.2 Stateful Load Balancer

A stateful or dynamic LB allows to keep track of the client session state by saving into a specific database an entry that is unique for each flow. When a packet arrives to the LB, it checks in the flow table (the “session state” database) if the client requesting for connection has already been registered in it or not. If a record is already stored in the flow table, the packet is assigned to the corresponding server. Otherwise, the packet belongs to a new flow and it is consequently sent to a server following the rules of the LB. The flow table can be considered as the central and persistent

"mechanism" which stores all the connections between clients and servers.

Hence, in a stateful LB, it is important to think about how the new connections are managed. It is necessary to decide how the requests could be distributed among the different servers, trying to avoid overload and to distribute the traffic as evenly as possible. However, the advantage that is obtained thanks to the optimization of load balancing involves a limitation in terms of performance. Stateful LBs are less efficient than stateless as they have more calculations to perform. While stateless only needs to compute the hash and send the packet accordingly, the stateful must first calculate the hash, secondly check the presence of the hash in the flow table and finally send the packet to the correct server. The flow table operations and its update require processing time, which is translated into higher latency and lower throughput or, in other words, lower performances. Moreover, a flow-table maintenance algorithm is required in order to remove the "expired" entries from it. Several algorithms can be used to perform this problem, and, in general, the computational cost of this operation is not negligible [12].

2.2 Data Plane Development Kit

The Data Plane Development Kit (DPDK) [7] is a set of drivers and libraries that allow to accelerate packet processing on a wide variety of NICs and CPU architectures. DPDK also provides two execution models [5, 7]: a pipeline model where one core takes the packets received and delivers them to another core that will process them, and a run-to-completion model that distributes packets to the different cores, processing and transmitting them at the same core.

2.3 Non-Uniform Memory Access

NUMA is the acronym for "Non-Uniform Memory Access" [8]. It is a type of memory design used in multiprocessor systems to achieve higher performances. With NUMA, different processors, in addition to their local memory and the shared memory, can also share each other's local memory. This comes with a cost: a processor can access faster its local memory with respect to the other processor's local memory and slower to the shared memory. Both the machines used in this work have a dual NUMA-node setup. Hence, if particular configurations are used, higher performance can be obtained with respect to a generic architecture. More details can be found in section 3.3.1.

2.4 Click and FastClick

Click is a modular and flexible software toolkit that can be used to create software routers [6]. However, several other network functions can be implemented through it. The main feature of Click is its modularity. In detail, each Click module is called "element", and it is responsible

for performing a specific task. For example, the module called DecIPTTL decreases the packet TTL and discards the packet when the time has expired. Several Click elements have already been implemented to perform the most relevant network functions, but it is also possible to create new elements to meet specific needs since the project is open source. To design an application (or, better, a set of network functionalities) using Click, it is necessary to create a configuration file where all the elements to be used and the interconnections between them are declared and configured. A detailed view of the structure will be given in chapter 4.

FastClick is based on Click and has added functionalities which allow it to be faster than Click itself. These functionalities are convenient computation of raw packet batches, multi-queuing support, and integration with DPDK [7]. In fact, it is exactly the integration with DPDK which allows the LB not to need system calls after every packet [5] as it skips the kernel network stack. In other words, it is the software on its own that retrieves the packets from the interfaces and deals with the delivery of them. This feature enables FastClick to process packets at a very high speed.

2.5 Hashing and collision in hash table

Hashing is a technique to uniquely identify a specific object from a group of similar objects by assigning a unique key to that particular element. Hashing can be used in applications when there is a need to store many elements and a linear accessing cost (like the one of an array) is not admissible. For this reason, special data structures, called hash tables, have been developed. In the specific case of the stateful LB, the main goal of hash tables is to make searching and insertion of hashed flow reference as easier and faster as possible. The hash function is used to compute an index that informs where an entry can be found or inserted. Sometimes it could happen that pairs of elements are mapped to the same hash value: this is called collision. To avoid collisions and to keep the data structure organized, hash tables need to implement some algorithm that, periodically, updates and maintains the table. In this way, a degrading of the performance due to high occupancy could be avoided.

2.6 Related work

In this section different flow classification methods have been taken into account and divided into two different groups: B-Tree methods and Hashing methods.

2.6.1 Binary Tree based flow classification methods

Tree-based flow classification methods are based on binary decision tree structures. A binary tree is a set of nodes connected with each other where a single node is assigned as root and each one

cannot have more than two “children”.

These methods start with a huge set of rules for flow classification which is then divided, or cut, into subsection of rules which, in turn, can be further split among the nodes. This means that given a set of rules, all of them are split in different nodes, but it is not necessary that one rule fully belonged to one particular node. In fact, it is possible, for a node, to have only a small percentage of a rule. This is exactly how behaves HiCut [13], one of the flow classification algorithms considered as one of the state-of-the-art methods.

One of the direct evolution of HiCut is HyperCuts [14]. Despite other tree based methods [15], HyperCuts has built-in functions which deal with replicated and empty elements, resulting in higher memory efficiency and lower search time when compared to HiCut. For example, it compacts the memory allocated for every single node so that there are no more empty spaces: that particular region will therefore include only the precise amount of memory covered by all the rules of that particular node. In addition, HyperCuts extends HiCut’s capabilities [14] by allowing multiple dimension cuts [16] at each node.

BitCuts [17] is another decision tree based algorithm which represents an improvement over HyperCuts. The depth of the tree produced by bit cuttings is smaller than other tree algorithms, resulting in an increase of search speed inside the decision tree. This leads to a much lower memory utilization and access. By using it into a DPDK environment, Liu Zhi et al. managed to double the throughput when compared with HyperCuts.

The problem with binary tree methods is that each time that a new flow is added, the entire tree needs to be modified. Considering that LBs receive many new flows [3] and that rebuilding the entire tree would significantly reduce the performance, these methods are not suitable for our purpose.

2.6.2 Hash based flow classification method

As previously stated in section 2.5, hashing is a process which generates an output with a fixed dimension starting from an input having a variable dimension. This can be achieved thanks to the usage of mathematical formulas also known as hash functions. One of the first methods which used a hashing algorithm to do the packet classification is Cuckoo hashing [18]. It maps each item to multiple candidate buckets using hash, then, the single item is stored in one of the buckets. The insertion of a new item may relocate existing items to their alternate candidate buckets. In order not to keep the insertion time too high, Cuckoo hash tables should not reach a utilization higher than 90% [19].

MemC3 [20] is an extension of Cuckoo hashing, which proposes optimistic concurrent Cuckoo hashing. This improves performance by allowing concurrent access to the hash table by multiple readers and a single writer, achieving over 90% space utilization. MemC3 employs Memcached [21] with CLOCK-based cache management [22]. Compared to original Memcached, it reduces the

space overhead by more than 20 Bytes per entry, improving the system throughput by three times while storing 30% more objects for small key-value pairs.

CUCKOOSWITCH [23] further improves the optimistic concurrent Cuckoo hashing method as proposed by MemC3 by adding x86 memory ordering, dynamic batching and prefetching. These optimizations improve the performance by approximately 5.6 over the original concurrent Cuckoo hashing.

Cuckoo++ [24] proposes an improvement of the Cuckoo hash tables by adding a bloom filter to each bucket. This implementation aims to decrease memory access while maintaining high performance. Compared to the optimistic approach (e.g. MemC3, CUCKOOSWITCH) and to DPDK, Cuckoo++ is evaluated to perform better during lookup both in single and multi-core environment. More details on this method are reported in section 4.2.6

2.6.3 Summary

We proposed a description of different methods which dealt with flow classification. Although, we chose not to implement HyperCuts or BitCut in Click because they are based on binary trees, which have a complexity of $O(\log(n))$. Even if hash tables have the worst cost of $O(n)$, their amortized cost is $O(1)$. Thus, on average, hash tables are more efficient than binary trees. As a consequence, thanks also to the (theoretically) better performance with respect to the other methods, we decided to implement Cuckoo++.

Chapter 3

Measurement technique

The purpose of this chapter is to provide an overview of the research method used in this work. Section 3.1 describes the research process. Section 3.2 details the research paradigm. Section 3.3 describes the experimental design. Section 3.4 focuses on the data collection techniques used for this research, while the section 3.5 is about the profiling of our application. Section 3.6 explains the techniques used to evaluate the reliability and validity of the data collected. Finally, Section 3.7 describes the method used for the data analysis.

3.1 Research Process

Two machines have been used as testbed, one for the traffic generation and the other for the LB (more details in section 3.3.2). The LB machine was used to forward in a balanced way the traffic generated (and received) from the other machine. Once the stateless Load Balancer has been developed, the performance tests of Click with and without DPDK were performed. Both iPerf3 [25] and TRex [26] were used to perform the tests and collect the data to analyse and obtain results. Afterwards, the stateful LB have developed and its performances have been tested with the same tools and in the same environment.

Then, the different multi-core techniques have been tested, with the goal of improving the performances. For this purpose, the main tool that we used has been the profiling, since it allowed us to know which functions were the ones using the majority of CPU time. After the implementation of every method or algorithm, the aforementioned tools have been used to see the improvements.

The final part of the research process consisted of a general review and comparison between the methods implemented by us and other existing methods that are currently used to solve LB problems.

3.2 Research Paradigm

The paradigm we used to complete this work is interpretative since our final goal is to provide rich evidence with credible and justifiable accounts (internal validity). Research process and findings can be easily replicated and results can be made use of by externals in other situations (external validity) [27].

3.3 Experimental design

3.3.1 Testbed

Before talking about the setup and the test environment, we have to introduce two terms that are widely used in the next sections to simplify the explanations. Since one server acts both as the source and destination of the traffic, it can be logically divided into two sub-machines called "generator" and "sink" respectively. The generator acts as a client of a real-world scenario, while the sink side acts as the server pool of it. This division has been obtained through Linux network namespace [28], and it is necessary to avoid loopback traffic between the server and client processes of the test software.

The testbed is composed of two almost identical server machines whose hardware is described in section 3.5.2. For the purposes of this work, one server (server-2) acts as a Load Balancer, while the other (server-1) generates and receives the traffic that travels across it, with the aforementioned sink and generator concept.

This type of setup is common when evaluating middleboxes, with a machine acting as a traffic generator and the other as a "DUT" (Device Under Testing). From the LB (server-2) perspective, the traffic generator (server-1) acts both as the "World" and as the "Server Farm". The "World" side (towards the generator) can be considered as the interface where all the external traffic arrives, while the "Server Farm" side (towards the sink) is the place where the traffic must be directed. Figure 3.1 give better understanding about the testbed setup.

3.3.2 Hardware and Software used

The hardware used for the testbed and the experiments is composed by two Dell servers running Ubuntu 18.04.3 LTS. Both servers are equipped with a dual socket configuration hosting two Intel(R) Xeon(R) CPU E5-2667 v3 @3.20GHz with 8 cores and 16 threads achieved through hyper-threading [29].

Each CPUs has 8 DIMMs, each one having 16 GBs memory size, for a total of 128 Gigabytes of DDR4 memory running at the frequency of 2133 MHz and working in quad channel mode.

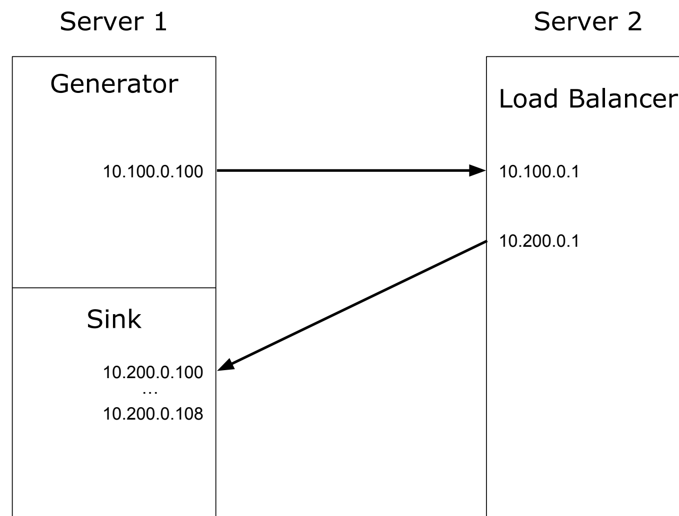


Figure 3.1: Testbed configuration

The network cards are two 100 Gb/s Mellanox Technologies MT27700 Family, model ConnectX-4 [30] on server-1 and one MT27800 Family, model ConnectX-5 [31] on server-2.

For what regards to the software part, we used FastClick in the LB, an improved version of Click that has additional functionalities such as FlowIPLoadBalancer (block structure and description are described in section 4.2). As stated in section 3.1, iPerf3 and TRex have been used to perform the measurements.

3.4 Data Collection

When evaluating the performances of a middlebox, like a LB, the most common technique is to use a traffic generator. This generates some dummy (but realistic) traffic on one of its interfaces, while it receives the traffic on the other one after going through the DUT, the LB in our case.

Initially, we used iPerf3 to test the performances, as described in section 3.3.1. To use this software, the network stack of the server has been split in two network namespaces [28] to avoid loopback traffic. Later, TRex has been used, as explained in section 3.3.2.

3.4.1 iPerf3 and the first round of tests

We started by testing the performance of the testbed and our configurations using iPerf3. iPerf3 is an open-source utility that generates one or more TCP/UDP flows. It allows very simple tests like throughput and latency tests between two hosts, with one host running it within client mode and the other one in server mode. However, one of its major limits is its intrinsic single thread structure, which limits the performance when dealing with high speeds. Furthermore, since we were also interested in managing a large number of connections, the structure of iPerf3 limited even more the results that we could have obtained.

To facilitate the tests, we developed some scripts that, through ssh, automatically started our Click application and iPerf3. Then, the JSON file output from the iPerf3 is analysed and, with the help of Python's MatPlotLib, the results are plotted.

We tried to run several iPerf3 instances together, in order to saturate the link or to put Click at its limits, but we did not manage to find a real bottleneck in the system. Several configurations have been tried, for example:

- 1, 4, 8, 16 iPerf3 instances with 1, 5, 10, 20 parallel streams for each instance
- different core bindings (with taskset, to take advantage of NUMA)
- different directions of flow (swap server/client instances)

The main issue that we found was the CPU load on the server-side of the test: since the CPU usage from these threads were always at 100%, we can not say if the performance was limited by the traffic generator setup (iPerf3) or from our Click configuration.

We have to admit that this was a productive failure, clearly showing that we can not use this setup to measure the performances of Click, and for the previously discussed reasons, we switched to TRex.

3.4.2 TRex

TRex is an open-source traffic generator software developed by Cisco, which can exploit the system features in a better way than iPerf3.

Apart from allowing massive traffic generation with a reasonable CPU usage and the creation of several different flows, TRex is able to use all the cores available on the server where it is running. Moreover, thanks to the DPDK native support, it can bypass the Linux Kernel network stack and achieve higher performances.

Due to the dual interfaces architecture installed on the server dedicated to traffic generation, a special configuration can be used to get the best performances. In particular, the client and server threads are bound to some specific CPU cores and, thanks to the NUMA architecture, this allowed us to avoid bus and memory limitations.

TRex can work in Stateless, Stateful and Advance Stateful mode. The difference between the last two modes is not clearly stated in the documentation, although, after some tests, we have seen that TRex in Stateless mode works correctly with our configuration and it can be used to stress the LB.

Thanks to this finding, we started testing the implementation of the LB through TRex in Stateless mode.

The tests have been performed with an automated script that used the Stateful client API of TRex to run a Stateless profile. In other words, the Stateful mode of TRex was used (to take advantage of the more extended APIs offered by it) with a Stateless test profile (to expoit the easier configuration and setup of it). Then,the results were collected and saved to the disk.

The tests have been run for 60 seconds each time, saving the throughput and latency data every second. TRex reported results have been saved every second, excluding the initial and final portion. In this way, we excluded the "adapt" phase of both TRex and Click, and the tail of the test, where TRex starts to kill the connections. These results are saved in a JSON format for future analysis. Sometimes, instabilities in the results have been noticed. In such situations, several more testing iterations have been performed in order to "amortize" the instabilities.

3.4.3 TRex result analysis

The results obtained in the above-mentioned method have been analysed through some Python scripts. These scripts are based on Pandas and Matplotlib to simplify the management of the data. To optimize the processing time that, due to the large amount of data saved, could become particularly long, the Pandas DataFrame were pre-computed and saved to disk (thanks to the Pickle library). In this way, the scripts did not need to reload the data from the JSON files every time the plots needed to be generated but could load the "cached" DataFrame.

To avoid the limits of the GitHub instance at KTH, the results have not been directly uploaded to the git repository but, rather, we uploaded only a tar.gz archive of the Pickle "cached" DataFrame. This allowed us to compress more than 1GB of raw data in some MB.

We can summarize the data processing chain as:

Individual JSON files (5.6GB) $\rightarrow$ `data.pickle` (2.3GB) $\rightarrow$ `data.pickle.tar.gz` (7.9MB)

We tried to re-use as much code as possible: the goal was to have few functions in a library that generated all the plots and, as a consequence, by editing such functions we could easily change the

aspect of all the plots, obtaining the maximum graphical consistency between all of them.

This strategy of files and functions organization allowed us to recreate all the plots starting from the results that we saved in the git repository, without copying different JSON files across different systems and without editing all the different plotting scripts but only the library.

In order to produce statistically-valid plots, we developed some functions that summarized and collected the data from several runs of click before plotting them. This allowed us to plot error bars and give an indication of how “stable” (or unstable) are the collected values.

3.4.4 Hacking TRex

While working with TRex, we faced the limits of its implementation of latency measurement. In particular, Trex measures the latency by a separate thread on different flows from the ones used for the “data” benchmark.

To calculate the latency, the latency packets are classified in some “buckets”: all the packets are classified according to their latency and counters about the latency are kept inside these buckets. However, to optimize the memory usage, the buckets are built on a logarithmic scale: starting with a resolution of 10 µs, the bucket “size” is increased by a factor of 10 after every 10 buckets.

For example, these are the first buckets used in TRex (values in µs):

```
10, 20, 30, ..., 80, 90, 100, 200, 300, ..., 800, 900, 1000, 2000, 3000, ...
```

To solve this problem, we have slightly modified the TRex source code making it save the latency data in 10 µs “buckets”, linearly. In this way, we had more information about the latency of all the packets and, more importantly, we could calculate a distribution function for the latency with a constant resolution.

Another change involved the JSON encoding. Instead of using a list of dictionaries, we used a simple array that, thanks to the fewer characters needed for each entry, resulted in smaller files.

For clarity, below we reported the old and new format of saving:

```
[{"key":10, "val":100}, {"key":20, "val":30}, {"key": 30, "val":19}]
```

Listing 3.1: Original TRex format

```
[100,30,19]
```

Listing 3.2: New saving format

This modification was particularly relevant due to the large amount of data that we needed to save every second: to avoid data loss, we saved the value of all the buckets up to 500 ms, resulting in 50000 values to be saved.

These modifications have been committed in a forked repository on github.com [32]. Several optimizations could be done in this field (like cutting away all the empty "buckets" after a series of empty buckets) or compressing the saved data. Future work could be done in this area and contribution to the original project could (and should) be submitted.

3.5 Profiling

A fundamental part in the evaluation of the performance of the LB is profiling.

Profiling consists of characterizing which functions are called at runtime, and which ones are the ones that occupy the CPU for the longest time. By obtaining this data, we can understand where the performances are lost and, then, we can think about how to improve and optimize the system.

To perform profiling, we used a Linux tool called perf [33]. Perf is capable to provide statistical profiling of the whole system but, in our case, we limited the monitoring only to the Click process. In particular, we ran the tests injecting packets in the system from the generator with TRex, we executed our Click LB application and after five seconds we started perf. The five seconds of waiting time before starting the profiling were needed to avoid recording possible initial performance instabilities.

The duration of the monitoring phase with perf lasted thirty seconds, at the end of which an output file with all the measurements were generated.

3.6 Assessing the reliability and validity of the data collected

The data collected are provided directly by the program responsible for the traffic generation, TRex, and perf. To guarantee valid and reliable data, we performed the tests multiple times, and then averaged all the results to obtain the final graphs and statistics. The high reliability and validity of the data is also secured by the high construct validity of the experiments. We ran the experiments managing all the possible variables and keeping them constant (e.g. CPU frequency, TRex traffic profile, Click configuration).

3.7 Planned Data Analysis

As also stated in the methodology section 1.4, several tests have been done for every different configuration. Our main features of interest during the acquisition of data were two: measurements, represented by throughput and latency (RTT and tail latency), and profiling, represented by the percentage of time that the CPU spends in each task. As a consequence, all the results that we obtained are quantitative results.

For what concerns the measurements, TRex allowed us to save the results in JSON format, then the results showed in the next chapters are plots which highlight the behavior of the LB using different numbers of cores in the computation.

For the profiling, screenshots have been taken to show the tasks in which the CPU spent the large majority of time. The reason for this choice is reported in the techniques that we used.

3.7.1 Data Analysis Technique

Plots are an essential part in our data analysis. We captured data from TRex in a standard JSON format, then we plotted the results according to our needs by a Python script.

For what regards perf, the software provides outputs in a particular format that, due to the "density" of the data collected, resulted in very big files (when using the `record` mode). Moreover, these files needed to be analysed on the same machine where the results have been collected to take advantage of the debug symbols (using the `report` mode). In order to reduce the amount of data collected and to speed-up the analysis of them, we decided not to save the results but, rather, to analyse them directly during run-time (via the `top` mode). The results here reported have been created by simply screen-shooting the output of perf. This is probably not the most efficient way, but we can consider it a good trade-off between precision of the results and time needed to produce them.

Chapter 4

Setup and Implementation

In this chapter, it is possible to find an explanation and description of our testbed setup and also the implementation of various LB scaling methods. In section 4.1 there are explained the steps to achieve the right setup of the testbed to be used for the LB implementations. Then, in section 4.2.1 there is a description of Click and FastClick elements that we use to build our LB. Continuing from section 4.2.2 to 4.2.6 we go into details of the implementation of the methods to scale the LB flow table using multiple cores, which is the core of this project.

4.1 Testbed setup

In order to perform our experiments, a setup of the machines was needed. Here, we have reported the main steps that we went through:

1. **Configuration of IPs and simple ping tests.** After this step both the machines had correct IPs which allowed them to have future connections and communications between each other.
2. **Configuration of network namespaces, routing rules and standard kernel-level routing.** As a result, both the "target" machines (sink and generator, as explained in [28]) were able to connect to each other using the IPs from the previous point. In this step, server-2 worked as a level 3 router.
3. **Replacement of Linux kernel router functionalities with FastClick.** The first FastClick implementations consisted of a basic router that replaced the kernel routing functionality of server-2. We first implemented it on top of the Linux network stack; then, we moved to DPDK, which allowed higher performances.

In particular, the point 3 can be seen as the first implementation step, since we created the Click configuration of a router, that is explained in detail in the next section and in Figure 4.1.

4.2 Implementation of the methods

To implement the LB methods, we started from the given Click [6] and FastClick [5] documentations and, after having deeply studied their behavior, we adapted it to our case. Therefore, based on the configuration shown in figure 4.1, we obtained a working router configuration. The scope of this task was mainly to familiarize with FastClick and its architecture.

Once we achieved the basic configuration (a router with and without DPDK support), we tested that the iPerf flow from generator to sink was going through our FastClick environment. Afterwards, we started the implementation of the stateless and stateful LBs.

Then, our work continued scaling on the multi-core environment of the Stateful LB. For this task, we modified the FastClick code of the *FlowIPManager* element in different ways as it is explained in section 4.2.2 and 4.2.4, and also changed the configuration of the *FlowIPManagerMP* as reported in 4.2.3.

In addition to this, in section 4.2.5 we also analysed the differences in terms of performance that occur when the computational part of the flow classification is moved from the CPU to the NIC.

The last part of our project consisted of the implementation of one new method to manage the flow table. This was selected after a literature review and then implemented. The chosen method is Cuckoo++ [24] and it is described in section 4.2.6.

4.2.1 Elements description

Click and FastClick provides several elements already implemented to perform network functions, such as network traffic classification, errors management, traffic routing, etc. The majority of these elements can be customized by changing the parameters in the configuration files. In this work, we also modified some existing FastClick elements and also coded new ones in order to achieve our goals.

In this section all the elements used in the configurations we created, both for a router and for the LB, are discussed and explained. Figure 4.1 and 4.2 provide understanding about how they are connected to each other.

4.2.1.1 Common elements

In the list below, the main common elements for both the router and LB implementations on FastClick are reported. A full description of all the elements can be found on the project Wiki [34].

Classifier
checks packet data according to a set of classifiers, one classifier per output port, then the packet is sent to the output port that matched.

ARPResponder
responds to ARP queries for IP addresses with the static Ethernet address.

ARPQuerier
encapsulates IP packets in Ethernet headers using ARP and asks for MAC addresses through the ARP protocol.

Strip
strips off first n bytes of each packet, where n is a parameter given in the configuration file.

CheckIPHeader
discards packets with invalid IP length, source address, or checksum fields.

GetIPAddress
copies the destination field from the IP header to the annotation.

DropBroadcasts
discards packets that arrived as link-level broadcasts.

IPGWOptions
processes IP Record Route and Timestamp options. Packets with invalid options go to second output.

DecIPTTL
decides if a packet's time-to-live (TTL) has expired. If it is, sends the packet on its second output. If the packet is still alive, decrements the TTL, updates the checksum.

IPFragmenter
fragments packets that larger than the configured Maximum Transmission Unit (MTU). Then sends the fragmented packets and packets smaller than MTU to first output, and sends the large packet with do not fragment flag to the second output.

ICMPError
encapsulates IP packets in ICMP error packets, sets Fix IP Source annotation.

The main difference between the router running without the DPDK support and the router running with DPDK is in the elements to receive and send the packets through the NIC. The router without

DPDK has the initial and last elements called *FromDevice* and *ToDevice*, whereas in the router with DPDK, those elements are *FromDPDKDevice* and *ToDPDKDevice* respectively. The *FromDevice/ToDevice* element acts as a sniffer by default, while *FromDPDKDevice/ToDPDKDevice* sets all received/sent packets to DPDK mode, being processed only once by skipping the kernel.

4.2.1.2 Router only elements

The below elements are used only in the router configuration:

Paint
: marks a packet with an integer "color". This can be used to "recognize" a packet later in the process chain.

LookupIPRouteMP
: decides the exit port of a packet by looking at its annotation. This is performed by using a static routing table.

CheckPaint
: emits every packet that matches the "paint" on its first output, while dropping all the others (or sending them to the second output, if it is connected).

4.2.1.3 LB only elements

In our work, we implemented LB only with DPDK support, so *FromDPDKDevice* and *ToDPDKDevice* elements have always been used to retrieve and send the packets. In addition, since the LB does not need to perform routing, the *LookupIPRouteMP* element is not needed. Furthermore, as it can be seen in figure 4.2, *Paint*, *CheckPaint* and *LookupIPRouteMP* are removed and the following elements are added:

IPLoadBalancer
: uses 5-tuples (IPs, ports and protocol) to hash the packets and selects one of the servers from the hashed value obtained. A list of the server pool IP needs to be specified as parameters in order to correctly send the packet to their destination.

IPLoadBalancerReverse
: sends the received packets from the different servers (sink) back to the generator. It needs the information of the *IPLoadBalancer*. Basically, it changes the destination IP of the packets to the LB's Virtual IP.

There is a major difference between stateless and stateful configurations: the stateful LB includes some additional elements described below:

FlowIPManager
: this element is responsible for managing the flow table; it is the place where the micro-flow state is stored. This element checks if a packet belongs to flow, and if it is the case of a new flow, it will add an entry to the flow table.

FlowIPLoadBalancer
: on the basis of the decisions performed by the *FlowIPManager*, the destination server IP is set as the destination of the packet.

FlowIPLoadBalancerReverse
: this element has the task of rewriting the packet header by changing the source. In fact, to be received correctly by the client, the packet must have the address of the LB as its source. For this reason in the return route there is no need of the FlowIPManager's table.

Figure 4.1 represents the diagram block of a router implemented in Click. To be more precise, it is the configuration proposed in [6].

Figure 4.2 shows our implementation of the stateful load balancer in FastClick.

The only difference between stateless and stateful LB in FastClick is that the former uses only *IPLoadbalancer* (and *IPLoadbalancerReverse*) without any element that manages and keeps the flow state, while the stateful LB uses *FlowIPLoadbalancer* (and *FlowIPLoadbalancerReverse*) combined with *FlowIPManager* that handle the flow table.

4.2.2 Global locked flow table

As soon as we realized that the bottleneck of our system was the LB, our task was to scale up the flow table using multiple cores to improve performance. Using FastClick, through the option "SCALE parallel" in the *FromDPDKDevice* element, it automatically scaled across cores specified when the program was launched.

The problem that occurred when scaling multi-core was similar to the one of concurrent programming: if a resource is shared among the processes, it must be ensured that only one process is accessing it at a specific moment or, in a less restrictive way, that all the data remain consistent for all the processes. Specifically, in our case the shared resource is the flow table element.

To solve this problem, different methods have been used, starting from the creation of a global locked flow table. Basically, to protect the table, we implemented a mechanism that blocks the access to all the threads with exception to the first that required the access. After every operation, the table is unlocked and another thread can have access to it. This mechanism is usually called Spinlock and it is a form of busy waiting. A more detailed description of it can be found in [35].

In FastClick, as already mentioned, the flow table is managed by the *FlowIPManager* element. To implement the lock we created a new element, *FlowIPManagerSpinlock*, which uses the Spin-

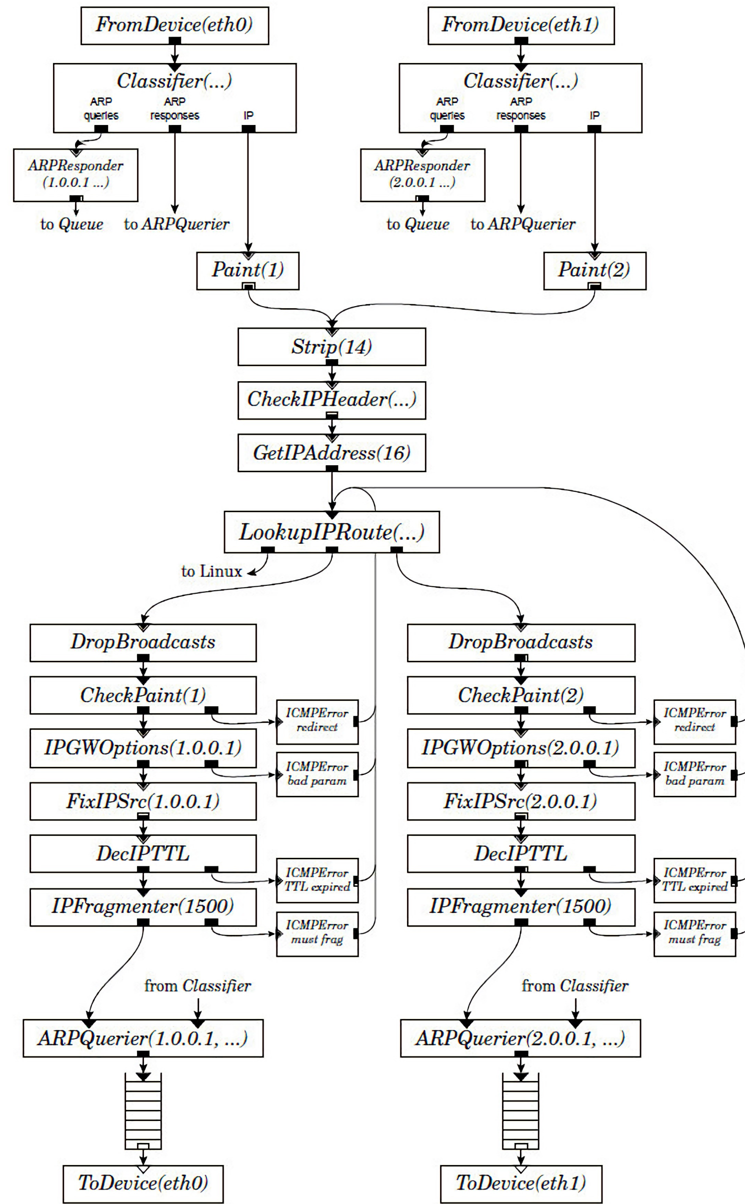


Figure 4.1: Click router configuration [6]

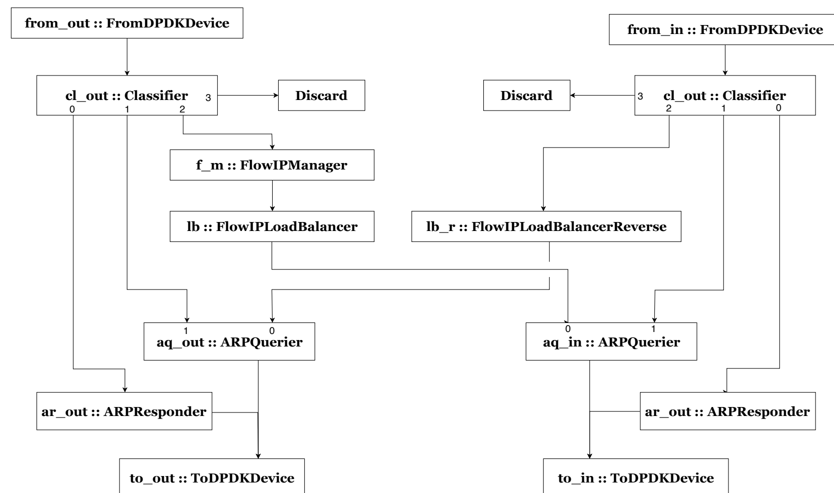


Figure 4.2: Stateful LB simplified configuration

lock class, already defined in Click. The lock is managed through the methods `acquire()` and `release()` of the SpinLock class.

In other words, the *FlowIPManagerSpinlock* is simply a *FlowIPManager* where there is a `name-of-spinlock.acquire()` instruction just before any operation on the flow table and a `name-of-spinlock.release()` instruction just after it. To use this element, a simple replacement of the *FlowIPManager* declaration with *FlowIPManagerSpinLock* was required.

As we were expecting and as reported in section 5.2.3, the performance with the Spinlock implementation are not good enough. This is due to the fact that many threads are in the waiting queue to access the flow table.

4.2.3 DPDK's Cuckoo hash table

DPDK uses bucketed hash table and the Cuckoo algorithm to avoid collision [7]. When performing the lookups, DPDK does it in batches following a pessimistic approach—prefetching both the primary and secondary buckets.

The FlowIPManager element relies on DPDK Cuckoo hash table [18] to map each input packet to the flow which it belongs to. Hence, all the choices taken by the LB can be done coherently for all the flows. In other words, this assures us that the packets of any given flows are always delivered to the same server. This element checks if a packet belongs to a flow and, in the case of a new flow, it adds an entry to the hash table. The Cuckoo hashing minimizes the collisions of the keys with respect to other simpler data structures.

The DPDK library already implements mechanisms to access the flow table in a thread-safe

manner. This mechanisms are controlled from some flags which determine the mode of operation of the hash table. Different flags (or different combinations of them) express different modes of operation. A list of the available flags is accessible in [36].

The flags are managed as binary masks. Encoded into an integer number, the various flags have the following values:

NAME OF THE FLAG	VALUE	ABBREVIATION
RTE_HASH_EXTRA_FLAGS_TRANS_MEM_SUPPORT	1	`TR`
RTE_HASH_EXTRA_FLAGS_MULTI_WRITER_ADD	2	`MW`
RTE_HASH_EXTRA_FLAGS_RW_CONCURRENCY	4	`RWC`
RTE_HASH_EXTRA_FLAGS_EXT_TABLE	8	-
RTE_HASH_EXTRA_FLAGS_NO_FREE_ON_DEL	16	-
RTE_HASH_EXTRA_FLAGS_RW_CONCURRENCY_LF	32	`LF`

Table 4.1: DPDK Cuckoo flags

As it is easy to understand, each flag has exactly one bit set. Consequently, by adding the decimal values or by doing a bitwise `OR` between the different flags we can combine them together. From this point, to simplify the reading of this document, we refer to the flags combination with their abbreviated representation. For example, we refer to the flag *RTE_HASH_EXTRA_FLAGS_MULTI_WRITER_ADD* as `MW`, following the nomenclature in Tab 4.1, while we use the *StF MW* when we are referring to the actual implementation of a stateful method with the `MW flag`.

FastClick already implements an element called *FlowIPManagerMP*, that accesses the table safely by setting the flags `MW` and `RWC`. In our work, we changed the combination of flags to understand the impact on the performances and the possible limitations. The flags that, from our tests, can have an impact on the performances are: `MW`, `RWC` and `LF`. The flag `TR` also impacts the performance in a negative way: since the machine used as LB does not support Transactional Memory, using `TR` flag in the DPDK's flow table implementation degrades the performances.

To perform the tests easier, we extended the *FlowIPManagerMP* element by allowing the user to set the flag value from the configuration file. This element can take both the decimal value or a list of all the flags to be set, for the maximum flexibility. Initially, we tested all the flags combination. After a brief analysis of the results obtained from the first tests, we selected combinations of flags that make more sense and tested them more into details. Results and discussion are reported in section 5.2.4 and section 5.5.

4.2.4 Per Core Duplication

We already discussed about the multi-core implementation of the *FlowIPManager* and the problem related to it in section 4.2.2. As we saw, having a unique flow table traversed by many threads

yields to a degradation on the performances.

To improve the performance in a multi-core environment, one possible solution is the creation of different flow table, one for each core (or, better, thread) used by the application. In this way, every process can read and write on his own flow table avoiding the concurrency problem.

This mechanism does not cause micro-flow coherence issues, since all the packets sent by the same client are always processed by the same CPU core, thanks to the RSS (Receive Side Scaling) feature of our NICs [37]. To be more specific, with RSS the NIC computes a hash for all incoming packets and, based on it, selects the CPU core that has to take care of each packet.

To implement the concept discussed above, we create a new FastClick module called *FlowIP-ManagerMP*. This element is responsible for initializing the flow tables and also for matching the thread with its own flow table during the processing of the packets.

As we reported in section 5.5, the performances are better than a Spinlock-protected flow table and, thus, this strategy could be effectively used in a multi-core environment.

4.2.5 Hardware-based classification

Until now, the LB application was in charged of hashing the packets and deciding which server was going to handle them.

Since the NICs used in our testbed have a built-in flow table, we wanted to exploit this feature and to delegate the task of keeping the flow track to the NICs, improving the overall performance. In this way, the LB would have been able focus only on changing the destinations of the packets and forwarding them.

The idea behind was the following: instead of hashing the packets of each flow that arrives to the LB, the NIC would have marked each packet of a new flow with the same number, following a rule that we set. Each time that we mark a packet, the action creates and sets to one two flags: `PKT_RX_FDIR` and `PKT_RX_FDIR_ID`. These flags will then be checked afterwards by the LB in order to do the forwarding.

The implementation was done by changing the source code of the *FlowIPManager* and adding the `rte_flow` API of DPDK. This API allowed us to use the NIC flow table and to use some actions such as MARK. By using this action, we have been able to set a unique identification number to each flow in order to distinguish between them.

The new FastClick module is called *FlowIPManagerHW* and it executes the tasks described. It uses the NIC flow table, marks packets from new flows and sets the corresponding flags up.

4.2.5.1 Hardware-only load balancer

Thanks to the APIs provided from the `rte_flow` library, another approach to the HW based classification could be taken.

Together with the MARK and FLAG actions, the network cards installed in our LB support different advanced traffic manipulation techniques. In particular, we were interested to the `SET_IPV4_SRC` [38] action, which would have allowed the NICs to change the destination IP of the packets matching the pattern to an arbitrary one. Obviously, by setting this IP to one of the servers' IP, we could perform the same tasks of a Load Balancer directly in the network cards. In this scenario, the LB functionalities degenerates to the insertion of the rules in the NICs whenever a new flow is recognized. In all the other cases (i.e. when a packet belonging to a known flow is received), the LB element simply forward the packet to the subsequent elements.

Thus, the *FlowIPManager* and the *IPLoadBalancer* could be replaced by a single block that, together with the NICs' rules, fulfils the same duties. The flow table and the FlowControlBlock data structures are not needed anymore, reducing the complexity of the software.

Another, more advanced setup could be envisioned by exporting all the LB functionalities to the NICs. The DPDK framework allows the user to configure the NICs to route the traffic between the different ports and different IPs. These functionalities are controlled both from the `rte_flow` library and from the traffic management APIs, with some examples provided in the documentation [39, 40]. In particular, the PHY_PORT, PORT_ID actions of `rte_flow` library should allow the interfaces to move the packets from one port to the other without letting them reaching the software application. This would be an ideal scenario, where the software (Click) would be needed only to setup and maintain the rules, leaving all the heavy job to the network hardware, ensuring the highest throughput and minimum latency possible.

During the last weeks of this work, we tried to implement the first idea, with the LB inserting the rules to change the destination IPs in the cards. However, we have not been able to obtain stable results and, thus, we cannot talk about the performances of this method.

Regarding the second, more advanced idea, we only briefly tested it due to the lack of time. Through the `rte_flow` APIs and the test console `testpmd`, we saw that, with the current setup, the needed instructions were not accepted by the card. From the DPDK documentation[41], these functionalities should be supported by the cards installed in the testbed machines. Hence, we believe that an additional software configuration may be needed. This can be a relevant starting point for some future works.

4.2.6 Cuckoo++

Bucketized Cuckoo hash-tables are heavily used in high-performance hash tables [42, 43, 44]. DPDK [7] utilizes them since they provide an easy and optimized way to prefetch the accessed memory.

Cuckoo++ changes the implementation of the Cuckoo hash table by adding a bloom filter to each bucket. The bloom filter contains the keys that do not figure in the first bucket so that it allows to determine whether a specific key is in the second bucket or not, without the need of accessing it. Contrary to the simultaneous prefetching of the primary and secondary buckets in DPDK, only the primary bucket is prefetched here.

If the key that is searched is not in the primary bucket then it is tested in the bloom filter. Either there is a negative answer, then there is no need to fetch the secondary bucket as the key is not there, or the answer is positive and in this case the fetching is done.

A new FastClick element has been created for the implementation of the Cuckoo++ flow table that replaces the DPDK's table. By using the header `rte_tch_hash.h`, from the library Cuckoo++[45], this element allows the selection of the flow table implementation at run time [24]. This can be done by passing a specific parameter (`IMPLEMENTATION`) to the Click element. In order to accommodate the new library functionalities, some of the code taken from the original *FlowIPManager* element has been modified, removing some functionalities not implemented in the library.

The final implementation is not multi-thread safe as the classification is performed using a unique Cuckoo++ hash table shared with the different threads. Hence, if this element is going to be used in a multi-core/multi-thread environment, a further implementation is needed. The *Per Core Duplication* introduced in section 4.2.4 may be a starting point for it.

Chapter 5

Results

In this chapter, after a detailed explanation about how we obtained and tested our testbed in section 5.1, the results regarding Throughput and Latency, together with the profiling are shown in section 5.2. Then, there is a discussion both in terms of Reliability 5.3 and Validity 5.4 ending with an overall general discussion in section 5.5

5.1 Testbed performances

In this section, the performance and behaviours of our testbed are reported. It goes through the system bottleneck and explains why the CPU clock is limited, analyzing the maximum throughput performance of the machines used and evaluates the latency.

5.1.1 System bottleneck

As we previously said in section 3.3.1 and 3.3.2, the system consists of two servers interconnected with two 100 Gbps links. However, in order to test and validate the improvements and the performance of our various configurations, it was necessary to define which were the performance of the hardware and in which scenario the performances were limited by the Load Balancer software. First of all, we investigated the CPU clock for server-2, where the LB software run. By running the software with the "standard" scheduler, at the maximum frequency, we noticed that Click is able to process more traffic than the received one and, consequently, we would have not been able to see the limitations (and improvements) of Click. The clock limitation could also be seen as an emulation of a commercial, modern network device: a lot of them are based on multi-core ARM architecture, which runs at a lower frequency (e.g. 1200MHz) but equipped with a large number of cores. To select an adequate frequency, we ran a test where we changed the number of cores and the clock frequency. The results of this test are reported in Figure 5.1.

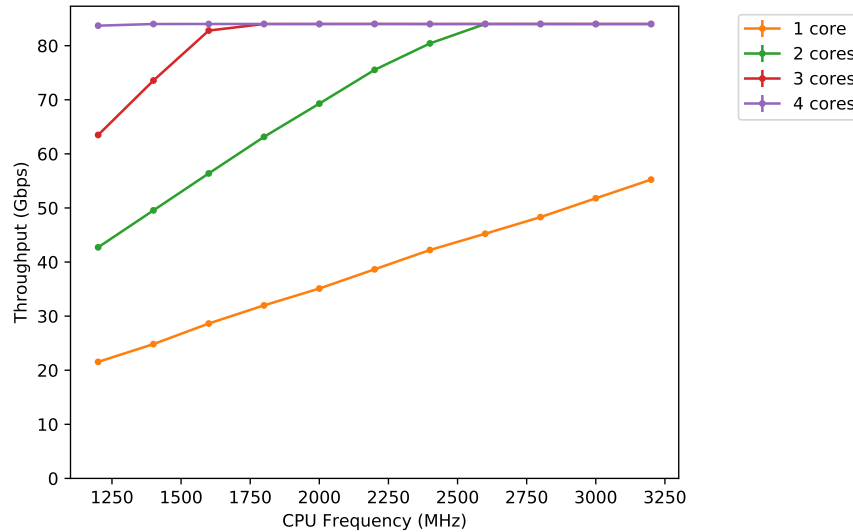
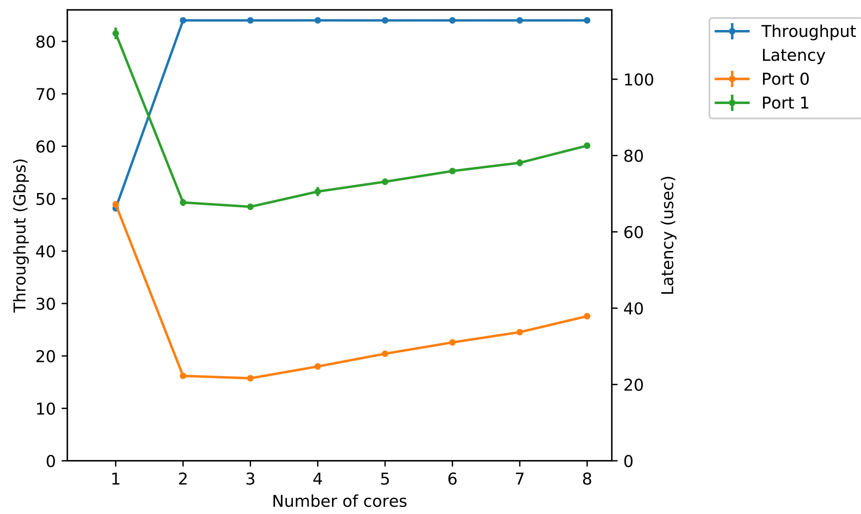


Figure 5.1: Throughput obtained with different number of cores and multiple frequencies configuration

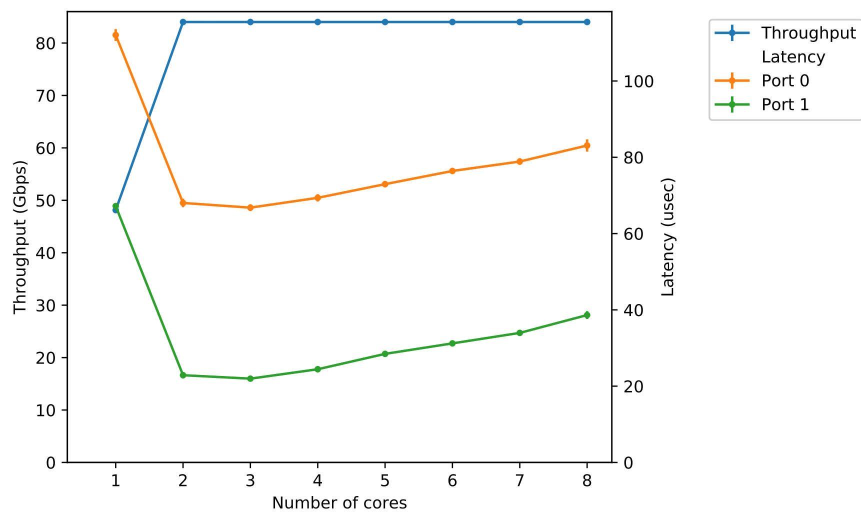
It is notable from the plot that for the higher frequencies our LB is not the bottleneck of the system but rather the performances are limited from the underlying hardware. In particular, the LB is not limiting the performance when more than 3 cores are used. The only way we have to ensure that our LB is the actual bottleneck is to force the execution with the lowest frequency available. Indeed, all the results reported from figure 5.2 have been obtained running test with a CPU clock forced at 1200 MHz.

5.1.2 Simple Forwarder Performance

As soon as we understood the right frequency to run the application, our goal was to find the physical limit of our testbed. To do this, we tested our system generating traffic at a rate greater than 100 Gbps and, by using a simple DPDK forwarder implemented in click, we looked at the throughput and latency of the system. The forwarder does not process the packets but simply forwards them to the output, leaving them unchanged, so by not adding computational steps, we can reasonably assume that the performance obtained with this configuration is the actual physical limit of our test environment. Figure 5.2 shows the throughput and latency performances obtained with the described configuration. As we can see the throughput limit is around 85 Gbps. Therefore, we can infer that throughput performance in the 83–86 Gbps range is limited by the testbed, while lower performances are due to the LB configuration.



(a) Standard setup



(b) Reverse setup

Figure 5.2: Throughput and Latency of the forwarder with different setups

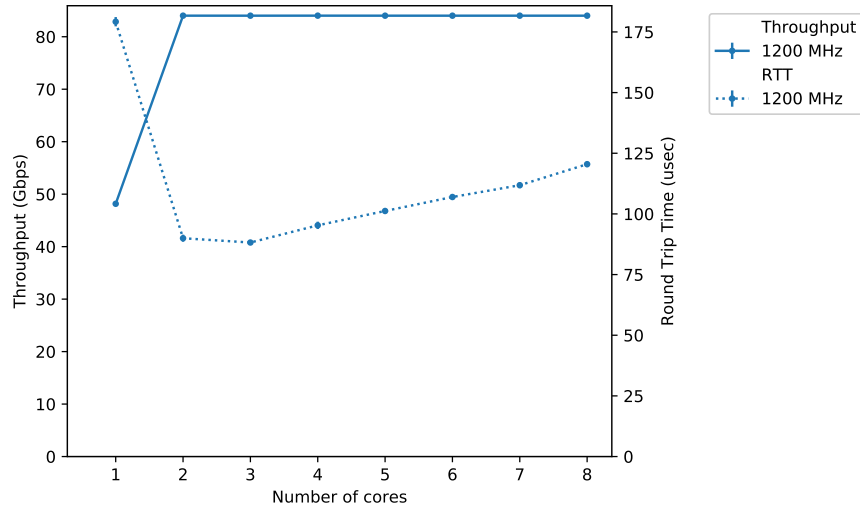


Figure 5.3: Throughput and RTT of the simple forwarder with different numbers of cores

5.1.3 Latency evaluation

Figure 5.2 shows that when the LB is no longer the bottleneck, the latency starts increasing, showing a significant difference between the latency of port 0 and port 1. These measures indicate the latency of the packets received on the expressed port. Namely, “latency port 0” describes the latency of packets sent from port 1 and received on port 0. Similarly, “latency port 1” indicates the packets sent from port 0 and collected on port 1. The most notable thing is that both latency lines have the same trend, but one of them has a higher latency. The difference between the two is constantly around 40 µs. To understand what this difference was due to, we performed the same test by changing the ports, and it is possible to see that between the two graphs in Figure 5.2, the port with higher latency has changed. A possible explanation that can justify this behavior is that TRex is running on multiple cores of the same CPU, but the two NICs are mapped through NUMA to different CPUs. This means that one NIC is exploiting NUMA while the other one does not and, consequently, one link performs better than the other. The justification of this claim is given by the fact that the difference between the two links is always constant, and it is similar to the delay introduced by the non-use of NUMA. In order to avoid report for every graph the two different latency, we calculate the RTT as the sum of the two latencies. Indeed, in the real world, the single way latency is not a meaningful value: it is usually more important to observe the Round Trip Time of the connections across the system. Thus, our choice of aggregating the two latencies is appropriate and represent a more meaningful value than the one-way latency. The performance of the forwarder in terms of throughput and RTT are reported in Figure 5.3.

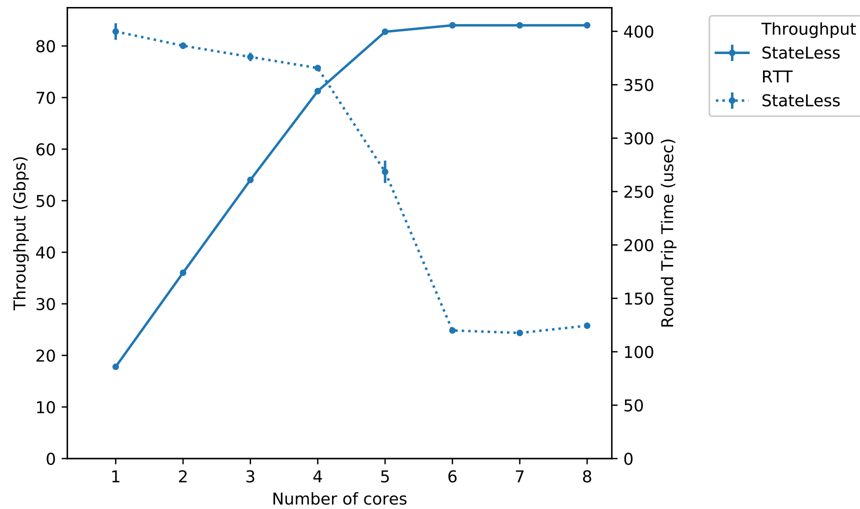


Figure 5.4: Throughput and RTT of the Stateless LB

5.2 Major results

In this section, the obtained results are reported. Every subsection is reporting the single result of a specific LB configuration. The discussion about the comparison between different methods is given in section 5.5.

5.2.1 Stateless LB performance

In this section, the throughput and latency performance results for the stateless LB are reported.

From the trends of throughput and latency reported in Figure 5.4 it is possible to observe how the LB is the bottleneck of our system as long as less than six cores are used.

To better understand what is happening in the LB, some profiling measures are reported in Figure 5.5.

It is easy to notice that in Figure 5.5a and 5.5b, the LB is the bottleneck since the CPU is used mainly by computation functions as *CheckIPHeader*, *click_in_cksum* and *ResetIPChecksum*, namely, functions of our interest. Figure 5.5c and 5.5d show that our LB is not anymore the bottleneck because most of the time is spent in functions regarding the NIC to read new packets (*mlx5_rx_burts_vec* and *FromDPDKDevice*). This is because Click is always using 100% of CPU and if it has nothing to process it is in polling to receive new packet from the network interfaces. Both the throughput/RTT graphs and the profiling results confirm that our LB stops being the bottleneck of the system when it is executed with six cores—if the 1200 MHz clock is set, as

```
Samples: 855K of event 'cycles:ppp', Event count (approx.): 19010880353
  Overhead  Shared Object        Symbol
+   13.47%  click                [.] click_in_cksum
+   10.54%  click                [.] ResetIPChecksum::simple_action
+    9.51%  click                [.] CheckIPHeader::simple_action_batch
+    9.39%  click                [.] mlx5_rx_burst_vec
+    6.70%  click                [.] mlx5_tx_burst_vec
```

(a) One core used

```
Samples: 621K of event 'cycles:ppp', Event count (approx.): 70332950111
  Overhead  Shared Object        Symbol
+   14.02%  click                [.] click_in_cksum
+   10.45%  click                [.] ResetIPChecksum::simple_action
+    9.65%  click                [.] CheckIPHeader::simple_action_batch
+    8.72%  click                [.] mlx5_rx_burst_vec
+    7.01%  click                [.] Classifier::push_batch
```

(b) Four cores used

```
Samples: 1M of event 'cycles:ppp', Event count (approx.): 111647378625
  Overhead  Shared Object        Symbol
+   12.53%  click                [.] mlx5_tx_burst_empw
+   11.56%  click                [.] click_in_cksum
+    9.32%  click                [.] mlx5_rx_burst_vec
+    8.80%  click                [.] ResetIPChecksum::simple_action
+    8.00%  click                [.] CheckIPHeader::simple_action_batch
```

(c) Six cores used

```
Samples: 1M of event 'cycles:ppp', Event count (approx.): 150026651410
  Overhead  Shared Object        Symbol
+   12.11%  click                [.] mlx5_rx_burst_vec
+    9.94%  click                [.] Classifier::push_batch
+    9.67%  click                [.] mlx5_tx_burst_empw
+    8.73%  click                [.] click_in_cksum
+    7.39%  click                [.] FromDPDKDevice::run_task
```

(d) Eight core used

Figure 5.5: Profiling of the Stateless LB

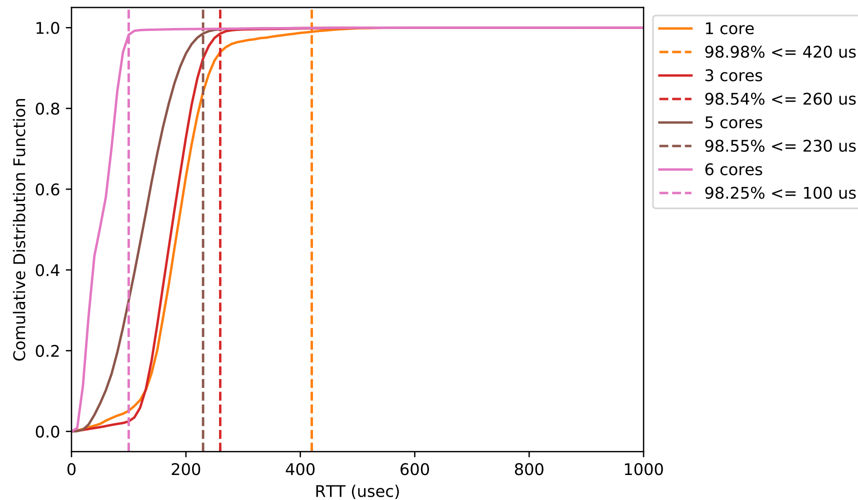


Figure 5.6: Cumulative Distribution Function of the RTT for the Stateless LB

previously mentioned. The CDF of the RTT for the stateless configuration is given in Figure 5.6 .

As expected, performance increased when the LB used more cores. We reported as an extreme the CDF with six cores, as the seven and eight cores plots where overlapping with it. When passing from three to five cores, the improvement of tail latency is only 30 µs, while passing from five to six cores we can see of a gap of 130 µs. We can justify this due to the fact that from five to six cores the LB is no longer the bottleneck of the system. Thus, in such a situation, the hardware platform introduces a constant delay for each packet, while with five cores or less, the packet delay is influenced only by the LB software.

5.2.2 Stateful LB performance

Our main goal was to characterize and evaluate different stateful LB methods and implementations. Since stateful LBs keep track of the flow to balance more efficiently the load, they require more computation with respect to the stateless ones. This explains the lower performances of stateful LBs.

The trend that can be seen in Figure 5.7 indicates a linear improvement in throughput based on the number of used cores. However, it is not the expected behavior for this implementation. In fact, with this configuration, the access to the flow table is not protected, therefore writing and reading can be performed simultaneously, generating errors. Unfortunately, the TRex mode used while performing these tests does not keep track of the flow states. Thus, TRex does not check if all the packets of one flow are always received from the same server. A related problem that may happen is related to the integrity of the table: if a thread accesses the flow table and writes in a cell where another thread is reading the destination of a given flow, its reading will be incoherent. The

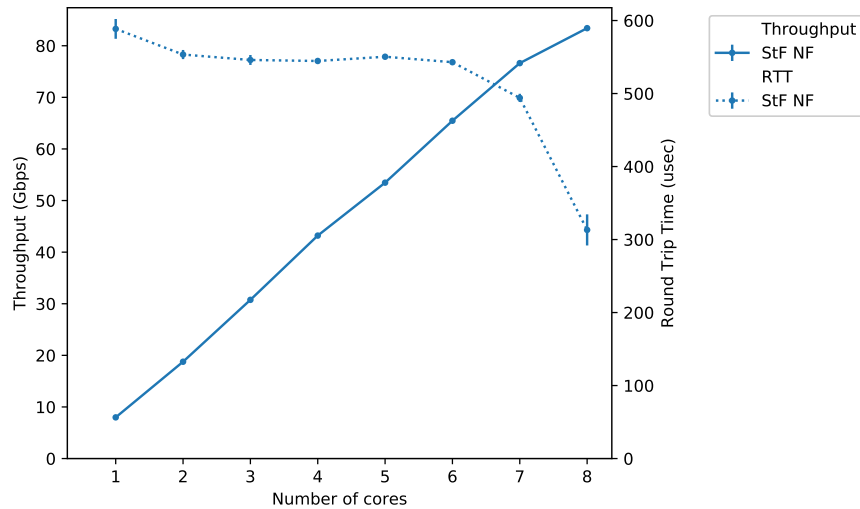


Figure 5.7: Throughput and RTT of the stateful LB with different numbers of cores

latter will then set the wrong destination for the flow, sending the packet to the wrong server. This packet, in a real scenario, should not be considered correct and consequently discarded. However, TRex, by not keeping track of source and destination, considers the packet as "good" and counts it as valid in the total measure of throughput. This is one of the limitations encountered during the work and, in a future work, another approach could be taken for testing these issues.

The profiling measures reported in Figure 5.8 help to see that using eight cores the LB is no longer the bottleneck of the system.

Lastly, Figure 5.9 is reporting the CDF of the RTT for the *Stateful* LB. It is clear that increasing the number of cores lead to a reduction in the RTT.

5.2.3 Stateful LB using global locked flow table

To improve the performance of the LB, one possible way is to scale up multi-core. *Stateful* LB uses the flow table to balance more efficiently the load, and the flow table can be considered as a resource where the process can write and read. For this reason when we are in a multi-core scenario we have to guarantee that the resource is safe, and in this case we managed it through a simple `spinLock`. The main problem with this solution is that the flow table is locked every time one process is reading or writing, and this implies that all the other processes have to wait for the resource to become free. The effect of this problem can be seen from the performances plotted in Figure 5.10 and also from the profiling in Figure 5.11.

From Figure 5.11a, it is clear that with only one core the LB does not have any concurrency problem. Indeed, the functions that take up more CPU time are the ones for packet processing

```
Samples: 250K of event 'cycles:ppp', Event count (approx.): 18641014658
  Overhead  Shared Object        Symbol
+   10.21%  click                [.] click_in_cksum
+    8.39%  click                [.] rte_hash_lookup
+    7.30%  click                [.] mlx5_rx_burst_vec
+    7.24%  click                [.] CheckIPHeader::simple_action_batch
+    6.25%  click                [.] ResetIPChecksum::simple_action
```

(a) One core used

```
Samples: 292K of event 'cycles:ppp', Event count (approx.): 59027425025
  Overhead  Shared Object        Symbol
+    9.97%  click                [.] click_in_cksum
+    9.68%  click                [.] mlx5_tx_burst_empw
+    7.78%  click                [.] ResetIPChecksum::simple_action
+    6.80%  click                [.] CheckIPHeader::simple_action_batch
+    6.04%  click                [.] mlx5_rx_burst_vec
```

(b) Five cores used

```
Samples: 297K of event 'cycles:ppp', Event count (approx.): 67470434132
  Overhead  Shared Object        Symbol
+   10.07%  click                [.] mlx5_tx_burst_empw
+    9.85%  click                [.] click_in_cksum
+    8.24%  click                [.] ResetIPChecksum::simple_action
+    6.72%  click                [.] CheckIPHeader::simple_action_batch
+    5.78%  click                [.] mlx5_rx_burst_vec
```

(c) Seven cores used

```
Samples: 677K of event 'cycles:ppp', Event count (approx.): 120937498092
  Overhead  Shared Object        Symbol
+    9.17%  click                [.] mlx5_tx_burst_empw
+    8.86%  click                [.] click_in_cksum
+    7.90%  click                [.] mlx5_rx_burst_vec
+    7.58%  click                [.] ResetIPChecksum::simple_action
+    6.12%  click                [.] CheckIPHeader::simple_action_batch
```

(d) Eight core used

Figure 5.8: Profiling of the *Stateful* LB

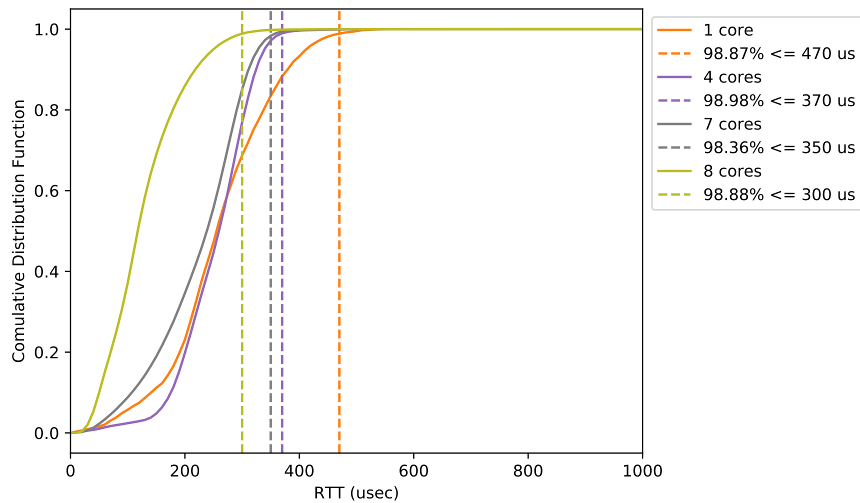


Figure 5.9: Cumulative Distribution Function of the RTT for the *Stateful* LB

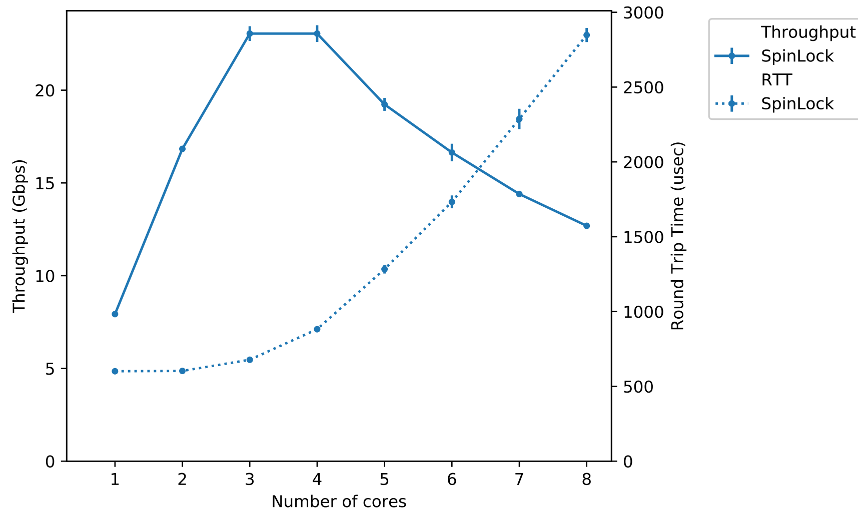


Figure 5.10: Throughput and RTT of the *Spinlock* LB

```
Samples: 25K of event 'cycles:ppp', Event count (approx.): 6445071076
  Overhead  Shared Object         Symbol
+   10.39%  click                 [.] click_in_cksum
+    8.84%  click                 [.] rte_hash_lookup
+    7.26%  click                 [.] CheckIPHeader::simple_action_batch
+    7.04%  click                 [.] mlx5_rx_burst_vec
+    5.57%  click                 [.] ResetIPChecksum::simple_action
```

(a) One core used

```
Samples: 132K of event 'cycles:ppp', Event count (approx.): 25182870676
  Overhead  Shared Object         Symbol
+   12.85%  click                 [.] FlowIPManagerSpinlock::push_batch
+    9.86%  click                 [.] click_in_cksum
+    6.87%  click                 [.] CheckIPHeader::simple_action_batch
+    6.22%  click                 [.] mlx5_rx_burst_vec
+    6.05%  click                 [.] ResetIPChecksum::simple_action
```

(b) Two cores used

```
Samples: 466K of event 'cycles:ppp', Event count (approx.): 76198689085
  Overhead  Shared Object         Symbol
+   54.94%  click                 [.] FlowIPManagerSpinlock::push_batch
+    4.70%  click                 [.] click_in_cksum
+    3.74%  click                 [.] mlx5_tx_burst_empw
+    3.21%  click                 [.] CheckIPHeader::simple_action_batch
+    3.09%  click                 [.] ResetIPChecksum::simple_action
```

(c) Five cores used

```
Samples: 631K of event 'cycles:ppp', Event count (approx.): 114216856457
  Overhead  Shared Object         Symbol
+   80.16%  click                 [.] FlowIPManagerSpinlock::push_batch
+    2.00%  click                 [.] click_in_cksum
+    1.54%  click                 [.] mlx5_tx_burst_empw
+    1.34%  click                 [.] CheckIPHeader::simple_action_batch
+    1.32%  click                 [.] rte_hash_lookup
```

(d) Eight core used

Figure 5.11: Profiling of the *Spinlock* LB

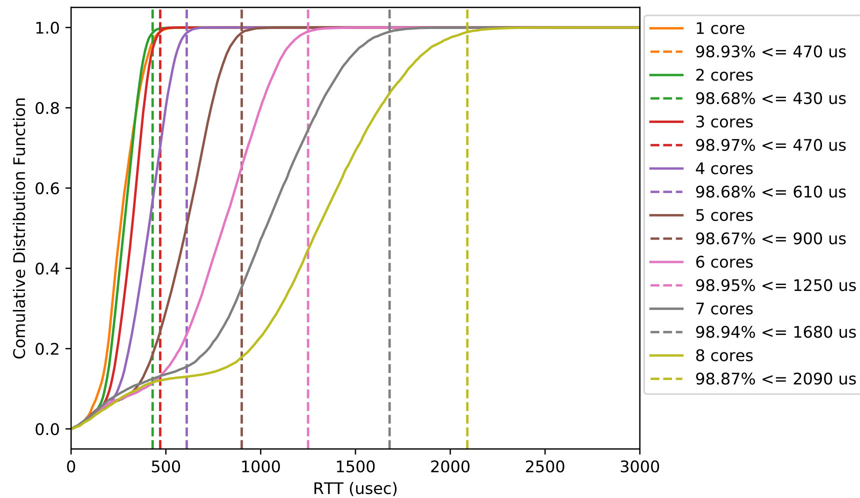


Figure 5.12: Cumulative Distribution Function of the RTT for the *Spinlock* LB

and lookup in the flow table. From figures 5.11b and 5.11d it is easy to notice that increasing the number of cores involved an increase of CPU time spent in one particular function. The concurrency problem is highlighted by *FlowIPManagerSpinlock* function, passing from 12% of CPU time with two cores to 80% with eight cores. This explains the degradation of the performance in terms of throughput and RTT. In other words, the different threads spend more time waiting the acquisition of the lock for the table than using it for the actual operations.

In Figure 5.12 the CDF of the RTT shows that with one up to three cores the best performances are obtained from the RTT point of view. The degradation of the performances is so high that it passes from having only 1% of the packet with a RTT higher than 430µs, to the scenario with eight cores where less than 20% have a lower RTT than 430µs.

5.2.4 Stateful LB using DPDK RTE_HASH_EXTRA_FLAGS

As already discussed in section 4.2.3, DPDK relies on a Cuckoo hash table. By setting specific flags, it is possible to obtain thread-safe access to the Cuckoo hash table. Since we expected that different combinations of flags could change the performance of the LB, we have tested different setups. The results are here reported.

5.2.4.1 `MULTI_WRITER_ADD`

The flag `MW` allows multiple threads to write on the table simultaneously. Readers are not protected from ongoing writes. This is basically the same behavior obtained with no flags, which is the

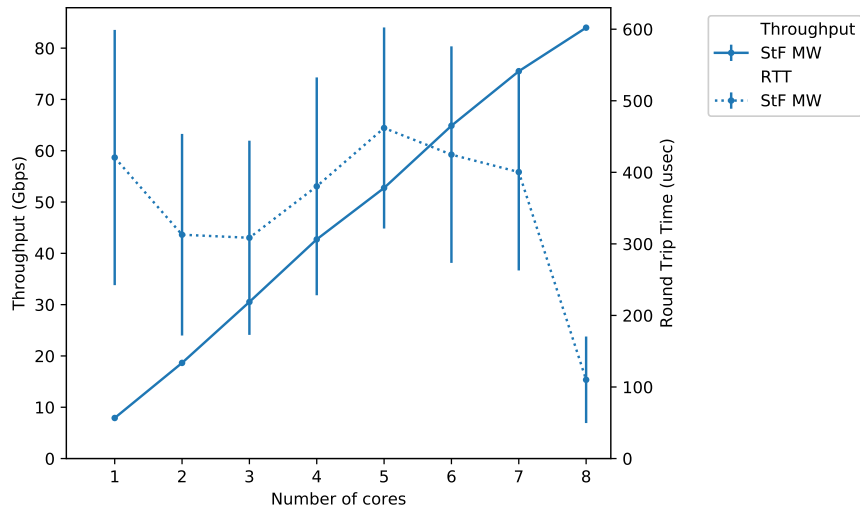


Figure 5.13: Throughput and RTT of Stateful with *MW* flag

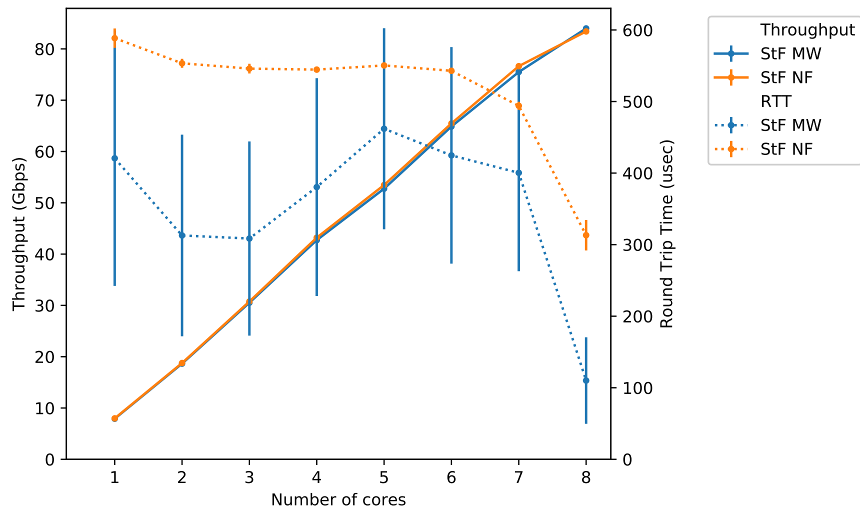


Figure 5.14: Comparison between *Stateful* (`Stf NF`, no flag set) and Stateful with `MW` flag

"normal" *Stateful* LB. Figure 5.13 shows a stable behavior for the throughput and a very unstable one for the RTT. Indeed, the error bar in the RTT trend shows high variance.

A comparison with the "normal" *Stateful* LB is reported in Figure 5.14, where it is clear that the throughput performance between the methods is the same, while there is a huge difference for what regards the RTT.

To better understand and give an explanation for the unstable RTT behavior, we reported the profiling result using different number of cores. From the profiling we did not notice any strange

behavior compared to the profiling of the `StF NF`, but we discovered that there is always an unstable RTT trend if the `MW` is set, as can be seen in and also in Figure 5.18.

The results in Figure 5.14 and Figure 5.18 show that the throughput remains the same, while using the `MW` flag the RTT is unstable even if on average it performs better than the configuration without the `MW` flag. Therefore, we can state that the cause of the instability of the RTT is caused by the `MW` flag utilization.

5.2.4.2 `RW_CONCURRENCY`

The flag `RWC` is responsible to guarantee safety in multi thread read and write operation. With this flag set, we expect to have lower performances than the simple *Stateful* LB and the configuration in section 5.2.4.1, since the access to the flow table will be secured. The performances are reported in Figure 5.16 .

As expected, increasing the number of cores involved a degradation in the performances. Up to four cores, the throughput performance is increasing, while the RTT is almost stable. With five cores or more, the performance begins to degrade. With eight cores the throughput performance is slightly worse and the RTT has doubled concerning the four cores scenario. Thus, assigning more than four cores, in this situation, does not improve the performances.

5.2.4.3 `MULTI_WRITER_ADD and RW_CONCURRENCY`

This is the combination of flags already implemented in the *FlowIPManagerMP* element on FastClick. This configuration allow multiple simultaneous writers (`MW`) but the access to the table is thread-safe thanks to the `RWC` flag. From now on, we refer to the *FlowIPManagerMP* as *StF MW RWC*.

Figure 5.17 shows that with this flags configuration the throughput is stable, but the variance of the RTT is still high. An important comparison could be done between this configuration and the one with only the `RWC` set. It is reported in Figure 5.18.

From Figure 5.18, we can derive that the throughput of *StF MW RWC* is comparable to the throughput of *StF RWC*, whereas the RTT of the former is always below the RTT of the latter. Thus, even if the *StF MW RWC* is more unstable concerning the RTT, it still performs better (or at least equivalently) than *StF RWC*. For this reason the *StF MW RWC* should be preferred to the *StF RWC*.

```
Samples: 25K of event 'cycles:ppp', Event count (approx.): 6433081498
  Overhead  Shared Object        Symbol
-    9.93%  click                [.] click_in_cksum
     9.93% click_in_cksum
-    8.98%  click                [.] rte_hash_lookup
   - 8.98% rte_hash_lookup
        6.18% 0
        1.88% rte_hash_k16_cmp_eq
+    7.02%  click                [.] mlx5_rx_burst_vec
+    6.98%  click                [.] CheckIPHeader::simple_action_batch
+    5.60%  click                [.] ResetIPChecksum::simple_action
```

(a) One core used

```
Samples: 227K of event 'cycles:ppp', Event count (approx.): 66515343911
  Overhead  Shared Object        Symbol
-   10.72%  click                [.] click_in_cksum
   - click_in_cksum
        CheckIPHeader::simple_action_batch
-    7.42%  click                [.] CheckIPHeader::simple_action_batch
     7.42% CheckIPHeader::simple_action_batch
-    7.18%  click                [.] ResetIPChecksum::simple_action
   + 7.18% ResetIPChecksum::simple_action
+    6.68%  click                [.] mlx5_rx_burst_vec
-    6.40%  click                [.] rte_hash_lookup
   - 6.40% rte_hash_lookup
        4.31% 0
        0.82% rte_hash_k16_cmp_eq
```

(b) Three cores used

```
Samples: 357K of event 'cycles:ppp', Event count (approx.): 76201896369
  Overhead  Shared Object        Symbol
-   10.26%  click                [.] mlx5_tx_burst_empw
   - 2.33% mlx5_tx_burst_empw
        3.54% ToDPDKDevice::flush_internal_tx_queue
-    9.99%  click                [.] click_in_cksum
   + 9.99% click_in_cksum
-    8.33%  click                [.] ResetIPChecksum::simple_action
   + 8.33% ResetIPChecksum::simple_action
+    6.74%  click                [.] CheckIPHeader::simple_action_batch
+    5.95%  click                [.] mlx5_rx_burst_vec
-    5.18%  click                [.] rte_hash_lookup
   - 5.18% rte_hash_lookup
        3.65% 0
```

(c) Seven cores used

```
Samples: 79K of event 'cycles:ppp', Event count (approx.): 22957098611
  Overhead  Shared Object        Symbol
-    9.03%  click                [.] mlx5_tx_burst_empw
   - 7.57% mlx5_tx_burst_empw
        8.54% ToDPDKDevice::flush_internal_tx_queue
-    8.94%  click                [.] click_in_cksum
   - click_in_cksum
        CheckIPHeader::simple_action_batch
-    8.03%  click                [.] mlx5_rx_burst_vec
   - 6.73% mlx5_rx_burst_vec
      + 7.96% FromDPDKDevice::run_task
+    7.43%  click                [.] ResetIPChecksum::simple_action
+    6.15%  click                [.] CheckIPHeader::simple_action_batch
+    5.69%  click                [.] Classifier::push_batch
+    4.48%  click                [.] FromDPDKDevice::run_task
+    4.47%  click                [.] ToDPDKDevice::push_batch
-    4.37%  click                [.] rte_hash_lookup
```

(d) Eight core used

Figure 5.15: Profiling of Stateful LB with `MW` flag set

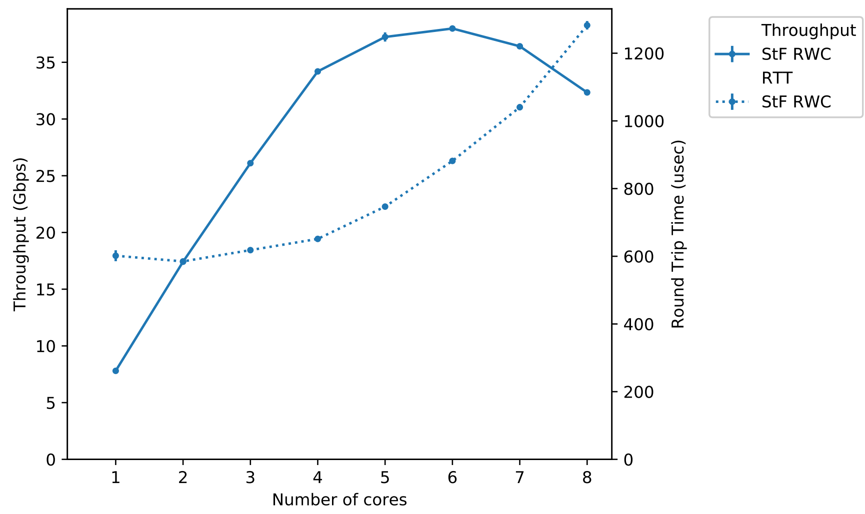


Figure 5.16: Throughput and RTT of Stateful with `RWC` flag

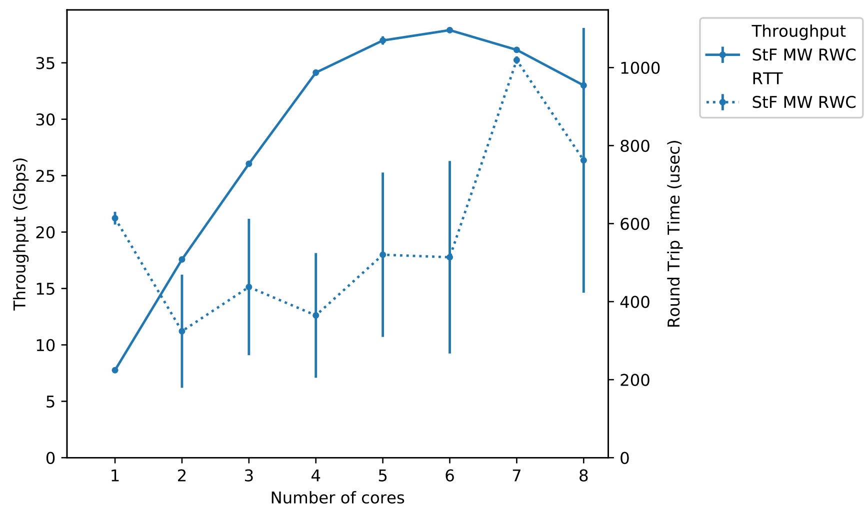


Figure 5.17: Throughput and RTT of Stateful with `StF MW RWC`

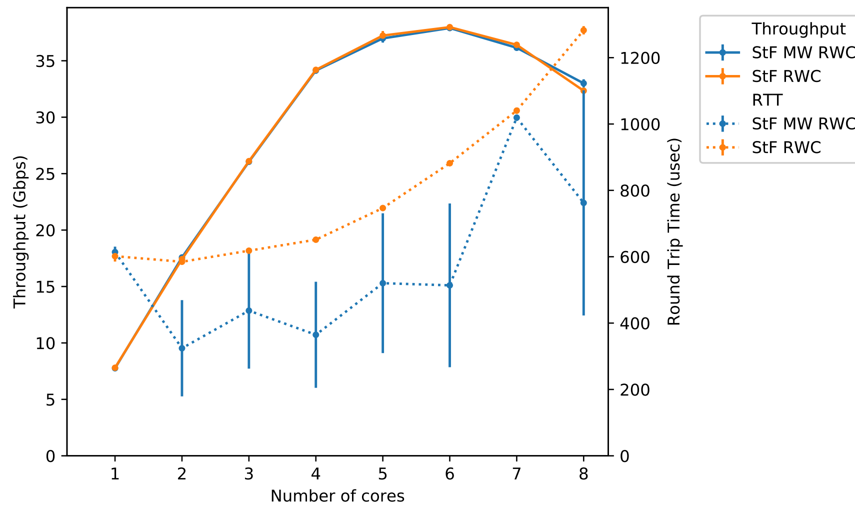


Figure 5.18: Comparison between `StF MW RWC` and `StF RWC`

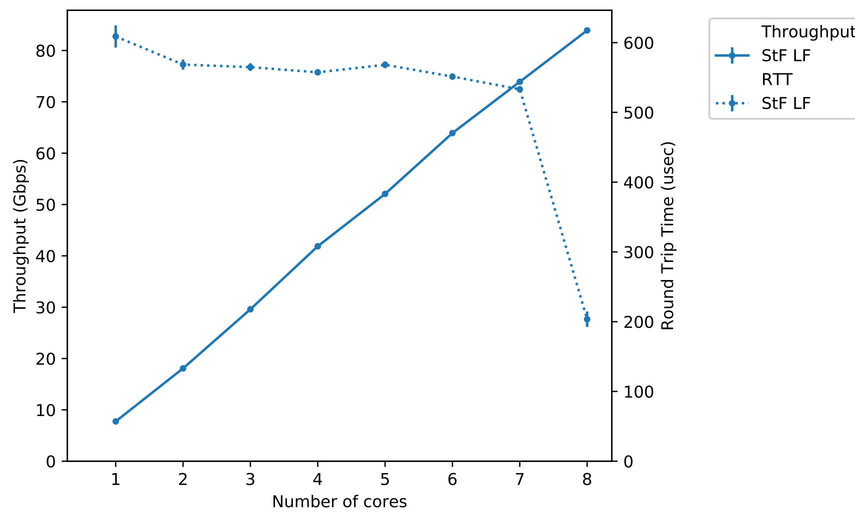


Figure 5.19: Throughput and RTT using `LF` flag

5.2.4.4 `RW_CONCURRENCY_LF`

The flag `LF` provides read and write concurrency without the need of a reader-writer lock. Authors of DPDK advise setting this flag to improve scalability in performance for platforms that do not support transactional memory, as our testbed. From Figure 5.19 the performances using the `LF` flag seems good, and a comparison with the other methods is provided in section 5.2.4.6

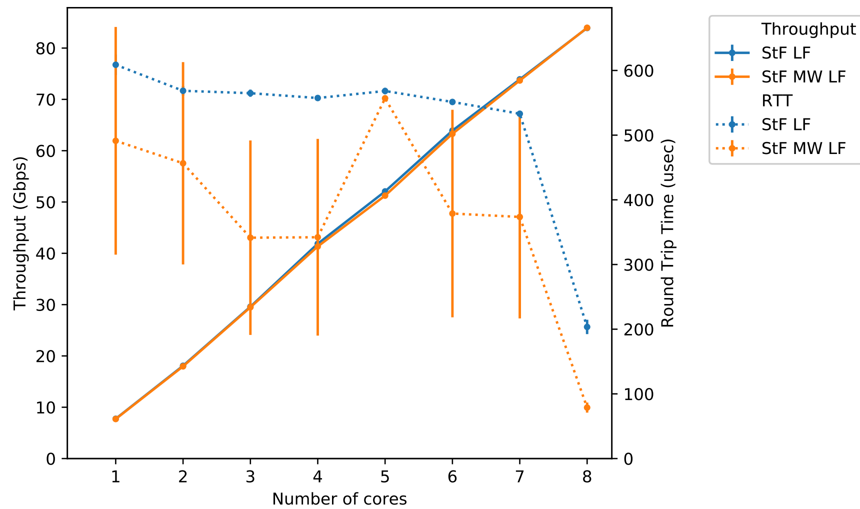


Figure 5.20: Comparison of Throughput and RTT of *StF MW LF* and *StF LF*

5.2.4.5 `MULTI_WRITER_ADD` and `RW_CONCURRENCY_LF`

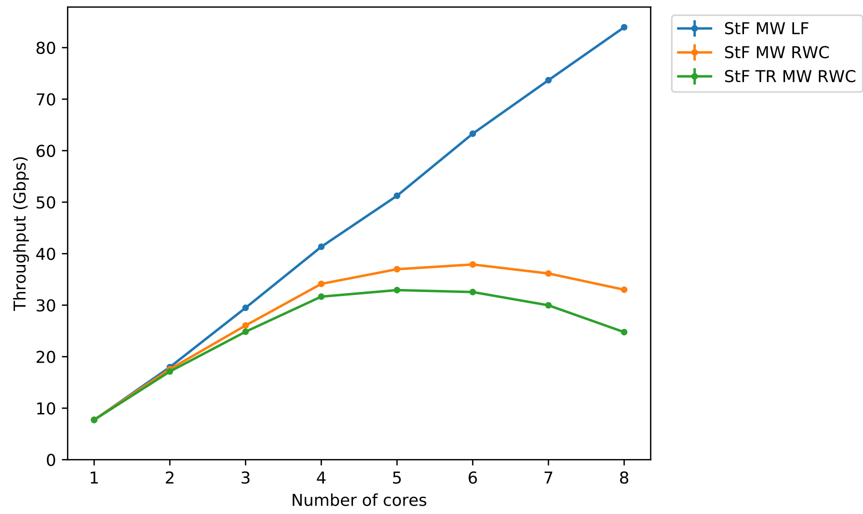
Here are reported the performances of a stateful configuration with both `LF` and `MW` set, called *StF MW LF*.

As already discussed in the section 5.2.4.1 the introduction of the `MW` flag involves a better performances but unstable behavior for the RTT. The results can be observed in Figure 5.20, where *StF MW LF* performances are compared to *StF LF*.

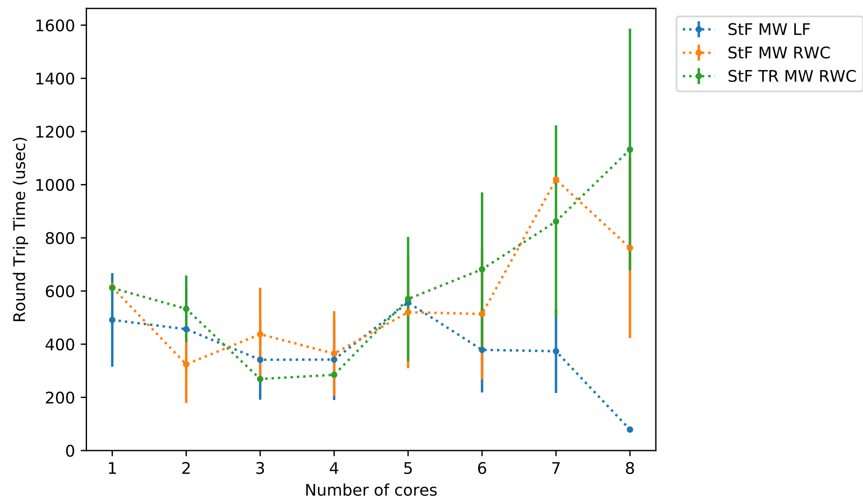
5.2.4.6 Comparison and consideration

We noticed from our experiments that changing the flags led to three main behaviors of performances. Figure 5.21 can help to have a better overview and a comparison of the trends in terms of throughput and RTT.

The worst performances are obtained when the flag for the transactional memory is set. The *StF MW RWC* works better, but the best performances are obtained with the `LF` flag. This is because it can guarantee thread safety without the use of a lock. The performances of the latter method are identical to those of the stateful with zero flag that, however, does not guarantee thread safety. Thus, from the results obtained we can affirm that in our system the combination of flags `LF` and `MW` is the one that performs better.



(a) Throughput



(b) RTT

Figure 5.21: Throughput and RTT comparison between different implementations

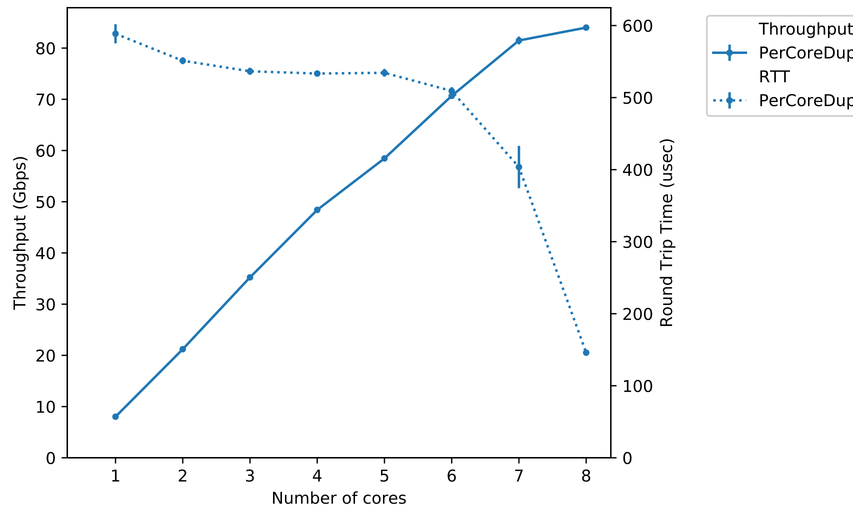


Figure 5.22: Throughput and RTT of the Per Core Duplication

5.2.5 Stateful LB using Per Core Duplication

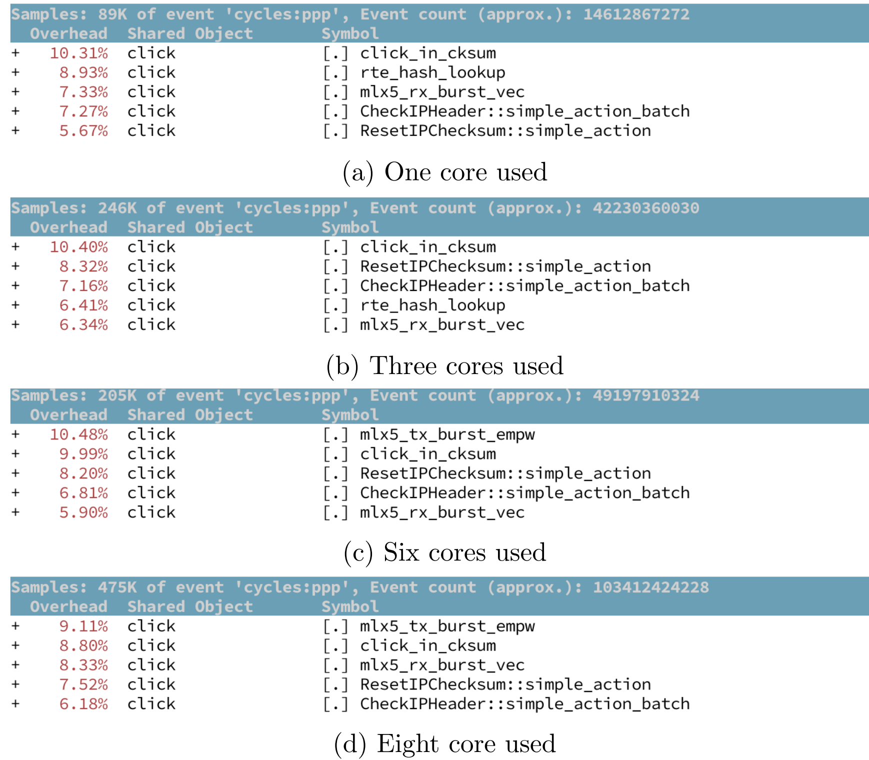
The Per Core Duplication is a method that consists of having a number of flow tables equal to the number of cores used. In this way we avoid the concurrency problem. Then, the performances are higher with respect to SpinLock, as we expected. Figure 5.22 report the throughput and latency trends for this configuration.

The profiling of the LB using the Per Core Duplication method does not show particular problems, as can be seen in Figure 5.23

Lastly, the CDF reported in Figure 5.24 shows clearly that — as expected — increasing the number of used cores implicates better performance on the RTT.

5.3 Reliability Analysis

All the results that we got, both in terms of throughput and RTT, have been obtained through TRex. During the test and the data analysis phase that we did, some strange RTT results have been noticed. A deep investigation of TRex made us discover that the latency evaluation system implemented is neither stable nor reliable enough. The most glaring example that justifies our statement is the instability we obtained with the MW flag. We indeed modified TRex to get some more information about latency, being able to plot the CDF for the RTT. In conclusion, the reliability of our results is based on the validity of the process used by TRex to provide the data.



```
Samples: 89K of event 'cycles:ppp', Event count (approx.): 14612867272
  Overhead  Shared Object        Symbol
+   10.31%  click                [.] click_in_cksum
+    8.93%  click                [.] rte_hash_lookup
+    7.33%  click                [.] mlx5_rx_burst_vec
+    7.27%  click                [.] CheckIPHeader::simple_action_batch
+    5.67%  click                [.] ResetIPChecksum::simple_action
```

(a) One core used

```
Samples: 246K of event 'cycles:ppp', Event count (approx.): 42230360030
  Overhead  Shared Object        Symbol
+   10.40%  click                [.] click_in_cksum
+    8.32%  click                [.] ResetIPChecksum::simple_action
+    7.16%  click                [.] CheckIPHeader::simple_action_batch
+    6.41%  click                [.] rte_hash_lookup
+    6.34%  click                [.] mlx5_rx_burst_vec
```

(b) Three cores used

```
Samples: 205K of event 'cycles:ppp', Event count (approx.): 49197910324
  Overhead  Shared Object        Symbol
+   10.48%  click                [.] mlx5_tx_burst_empw
+    9.99%  click                [.] click_in_cksum
+    8.20%  click                [.] ResetIPChecksum::simple_action
+    6.81%  click                [.] CheckIPHeader::simple_action_batch
+    5.90%  click                [.] mlx5_rx_burst_vec
```

(c) Six cores used

```
Samples: 475K of event 'cycles:ppp', Event count (approx.): 103412424228
  Overhead  Shared Object        Symbol
+    9.11%  click                [.] mlx5_tx_burst_empw
+    8.80%  click                [.] click_in_cksum
+    8.33%  click                [.] mlx5_rx_burst_vec
+    7.52%  click                [.] ResetIPChecksum::simple_action
+    6.18%  click                [.] CheckIPHeader::simple_action_batch
```

(d) Eight core used

Figure 5.23: Profiling of Stateful using *Per Core Duplication* method

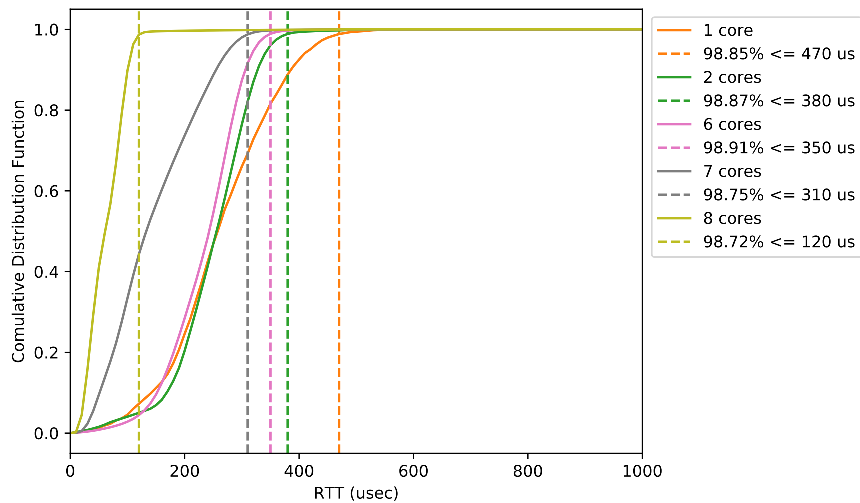


Figure 5.24: Cumulative Distribution Function of the RTT for the *Per Core Duplication* method

5.4 Validity Analysis

To have the highest possible internal validity, we repeated the tests multiple times, trying to control all the relevant background variables to create the same environment for every test. In particular, we ran the tests with a fixed CPU clock and always injecting the same amount of traffic with the same traffic profile on the network. All the results reported are the mean of five tests for each configuration, and as it can be noticed from the error bars (almost always absent) the results are stable. We can consequently claim that the results we obtained are valid. Regarding the results obtained with the implementation of *Cuckoo++*, we cannot say that they are valid. We have too few data available that only allowed us a general analysis. As already discussed in the section 4.2.6, the results cannot be considered 100% reliable, and further investigation on this topic is needed in future.

5.5 Discussion

In this section the results are discussed and the comparison between the different methods is given. In Figure 5.25 is reported the throughput behavior of different Stateful LB methods and Stateless. It shows that Stateless is the fastest method, since it has fewer operations to perform. However, as already stated above, it's not the optimal way to divide the load on the different servers.

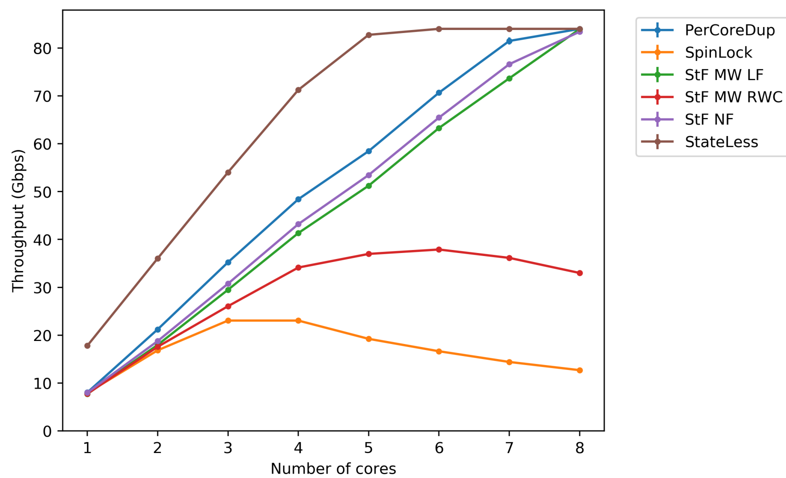
On the other hand, *Spinlock* is the method that has the worst performances due to the CPU contention. Since using more cores means a consequent increase of the CPU contention, with more cores the *Spinlock* performance is even worse. Thus, it can not be used in a real scenario.

The behavior of the *StF NF* was not expected. Indeed, without safe access to the flow table, we expected to see a crash of the application; instead, it showed better performance since there is no CPU contention. However, due to the non thread-safety of this method, it cannot be used in a real environment without a protection mechanism.

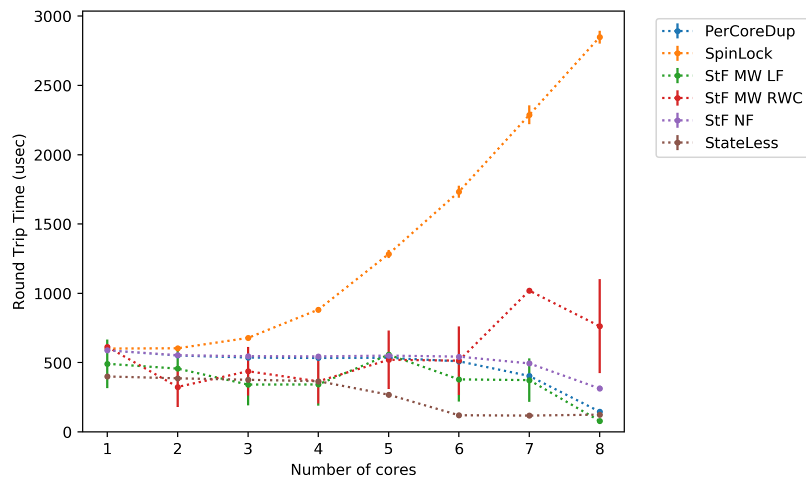
The *StF MW RCW* achieved better performances when compared to the *Spinlock*, but it also showed a similar pattern of performance degradation when increasing the number of cores.

The *StF MW LF* ensures thread safety with high throughput performance, comparable with the `StF NF` implementation. The strength of the *StF MW LF* is RTT performance, which is on average the best for stateful methods, and in some scenarios even better than Stateless. However, the RTT is unstable, due to the use of the `MW` flag

Finally, the *Per Core Duplication* turned out to be the stateful method that guarantees the highest throughput. The highest performances have been obtained because since each core has its own table and so there is no shared resource. Thus, no contention is needed and concurrency problems are avoided.



(a) Throughput



(b) RTT

Figure 5.25: Throughput and RTT comparison between Stateful methods

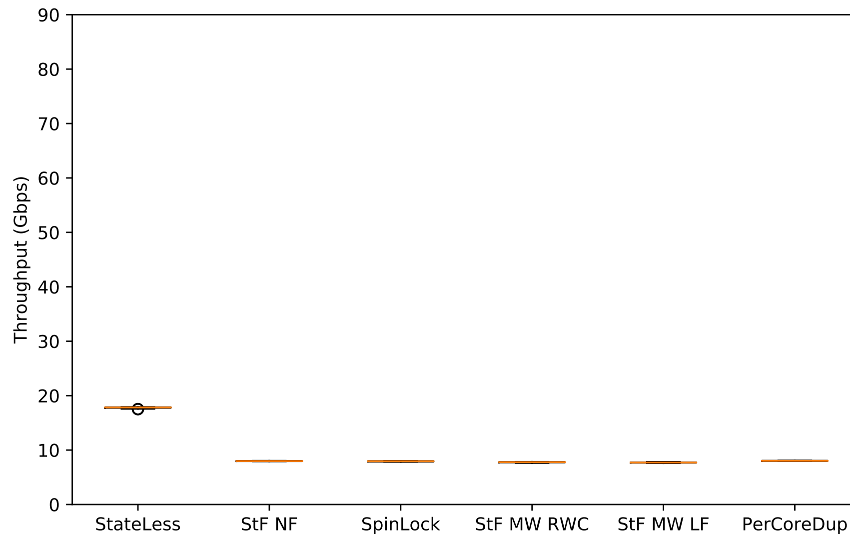
In conclusion, the *Per Core Duplication* and *StF MW LF* methods are comparable and can both be used in a real scenario. The choice between them relies on the different implementation: if a global, unique, table is needed (for example, to take decisions that needs a global knowledge), the *StF MW LF* method is the candidate. Otherwise, the *Per Core Duplication* is another reliable solution but, since it does not share the flow table across the different threads, it may restrict the number of parameters that a LB should consider when choosing the servers.

To get another perspective on the results, Figure from 5.26 to 5.29 report the distribution of throughput values and RTT through whiskers plot. Figure 5.26 reports the throughput and RTT distribution using one core. It can be noticed that throughput and RTT performance of the stateful methods on average are similar, while stateless performs better. When we start to scale up on an environment with multi-core, as shown in figures 5.27, 5.28 and 5.29, we notice the difference in performance between the various stateful methods. In particular, as already discussed, *Spinlock* is the slowest and worst method when increasing the number of cores. It can also be noted from Figure 5.29 that with eight cores *StF NF*, *StF MW LF* and *Per Core Duplication* reach the limit of the testbed. The final consideration that we have from the figure 5.29 is that when the LB is no longer the bottleneck of the system, the configuration *StF MW LF* guarantee an RTT performance even better than the Stateless.

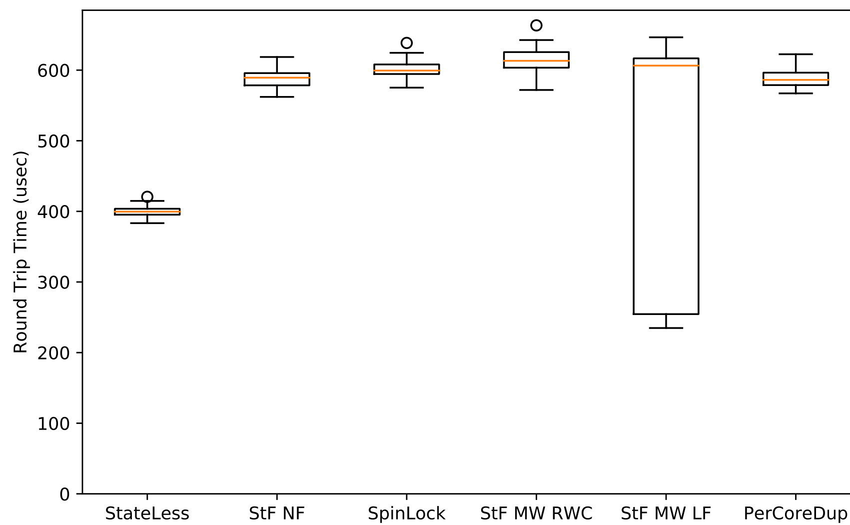
Figure 5.27 report the distribution executing the LB with three cores. In this scenario begin to be marked differences between the various methods. The *Per Core Duplication* is the stateful method that provides the best throughput. The interesting result is that the stateful with the *StF MW LF* reach RTT performance on average even better than the Stateless, even if the variance of *StF MW LF* is very high, while the Stateless is very stable. The *Spinlock* starts to be the worst method both for throughput and RTT.

Figure 5.28 reports the results with six cores. The aspect that can be noticed immediately is the degradation in the *Spinlock* performance. Regarding the Stateful methods, even in this case the Per Core Duplication is the best for throughput, while *StF MW LF* reach the lowest RTT. A notable thing is that in this case *Stateless* is overall the best method both in terms of throughput and RTT.

Figure 5.29 shows the result obtained with eight cores. A plot for the RTT without the *Spinlock* performances is also added to better understand the results for the other methods.

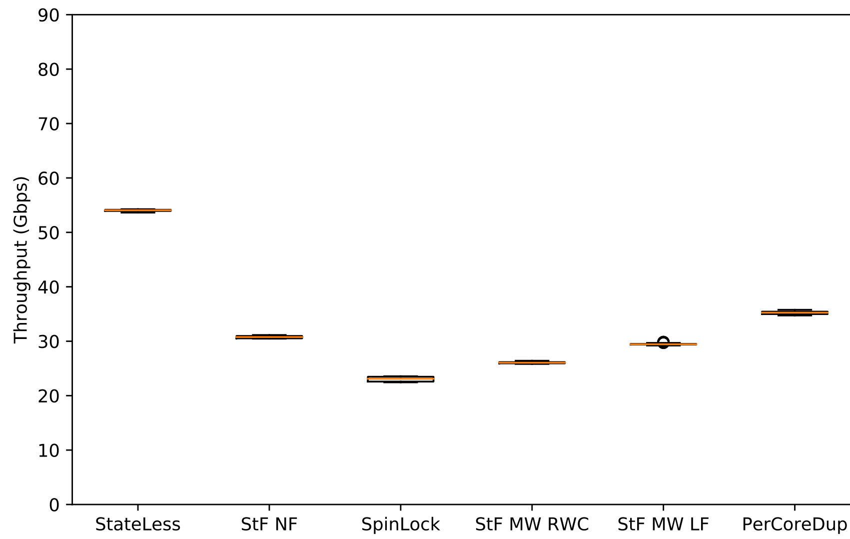


(a) Throughput

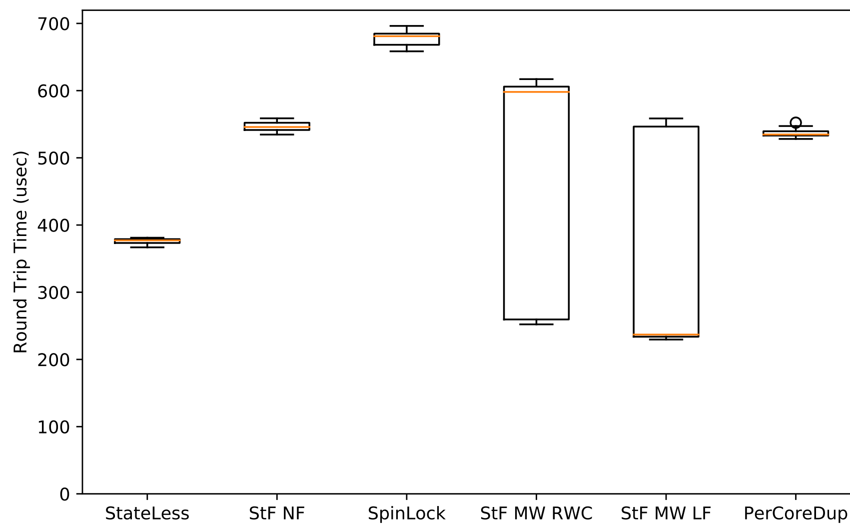


(b) RTT

Figure 5.26: Throughput and RTT comparison running with one core

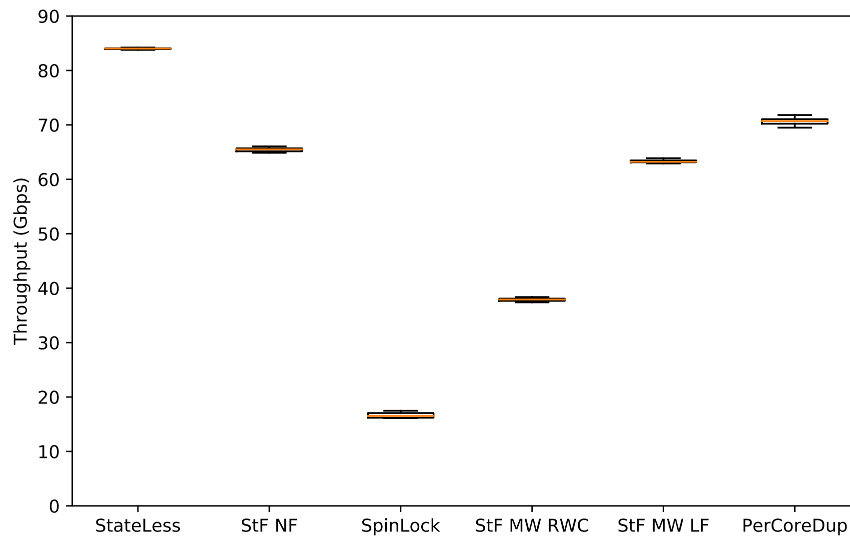


(a) Throughput

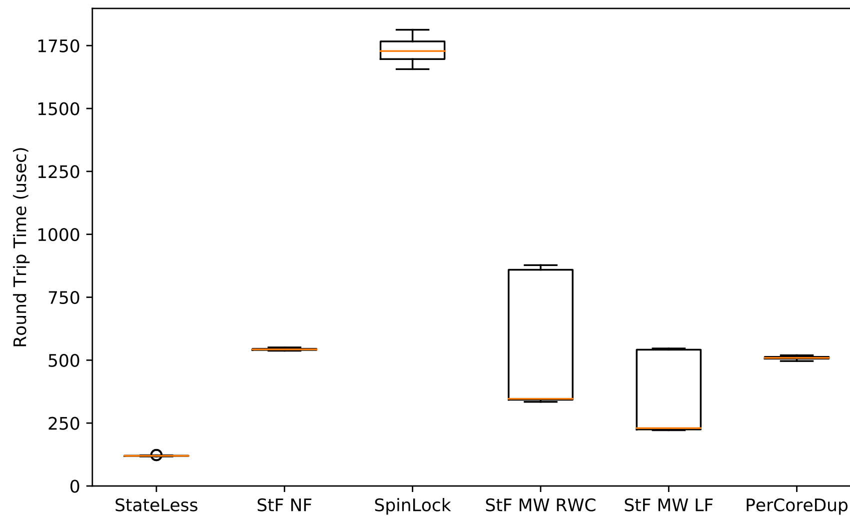


(b) RTT

Figure 5.27: Throughput and RTT comparison running with three cores

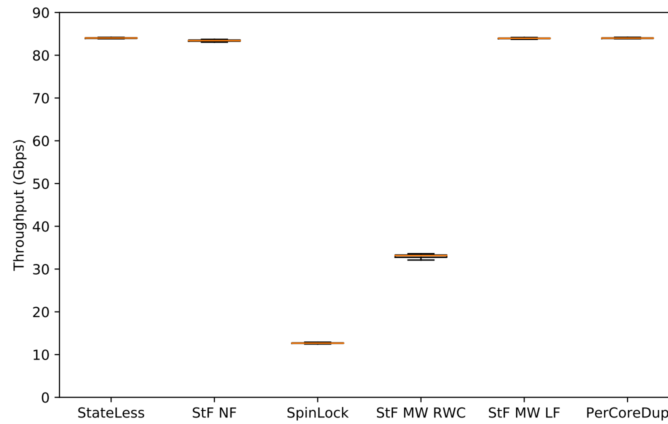


(a) Throughput

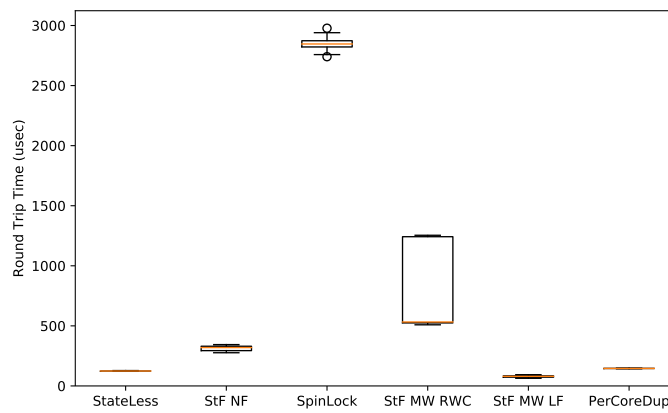


(b) RTT

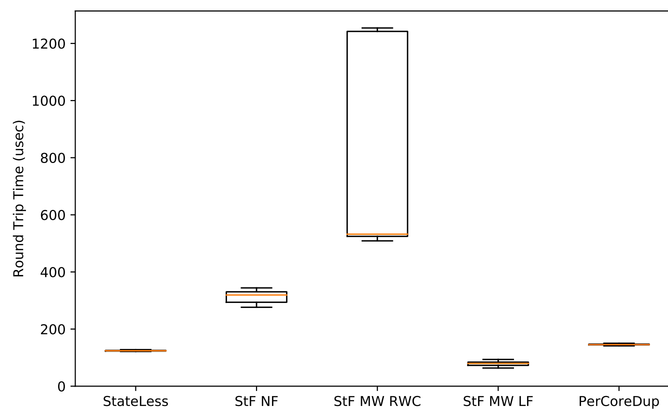
Figure 5.28: Throughput and RTT comparison running with six cores



(a) Throughput



(b) RTT



(c) RTT without the SpinLock

Figure 5.29: Throughput and RTT comparison running with eight cores

5.6 Hardware classification and Cuckoo++ results

In our work, we did not succeeded to obtain meaningful and valid results for two different implementations: HW classification and Cuckoo++. In this section, it is possible to find a brief explanation about the reasons behind this fact.

5.6.1 Hardware classification

Regarding HW based classification, we have not been able to obtain meaningful results due to implementation problems. Although we tried two different approaches as described in section 4.2.5 and 4.2.5.1, only some flows were correctly recognized following the rules by the network card. This led to performance in the order of Mbps, which are very far from those that we obtained with the other stateful methods.

5.6.2 Cuckoo++

For what concerns Cuckoo++, the results obtained cannot be considered valid since few tests have been performed and only in a single-core environment. This was due to the lack of time to carry out more in-depth and detailed tests in a multi-core environment. The results obtained in a single-core environment are comparable with those provided by the standard DPDK implementation (10 Gbps), but it should be considered that the code of Cuckoo++ has not been optimized yet. For this reason, thanks also to the outcomes obtained by the authors in [24], we believe that Cuckoo++ can be a valid choice for the management of the hash table. Therefore, a future and more extensive research could lead to interesting results.

Chapter 6

Conclusions and Future work

In this section we will expose the conclusions of our work, the limitations found, future work to be done and finally the reflections.

6.1 Conclusions

Our work researched how different load balancing methods affect the throughput and latency of a LB. We developed, tested and compared them, getting significant results and measures, being able to understand which ones perform better and why.

When comparing the performances of stateless and stateful LB methods, the stateless one has an overall better throughput and lower RTT as we could expect because it has fewer operations to perform. However, our work focused more on stateful methods and not stateless LBs, as the second ones do not keep the information of the flows.

As already discussed in section 5.5, the *Spinlock* method has the worst performance registered with our tests, as the CPU contention is higher when more cores are used. As the CPU contention increases, the RTT grows significantly and the throughput decreases if we use more than 4 cores, meaning that even though it provides a tread-safe environment it is not a convenient load balancing method.

The Per Core Duplication method should be privileged if higher throughput is seeked. On the other hand, the flags of DPDK's flow tables *MULTI_WRITER_ADD + RW_CONCURRENCY_LF* have to be preferred if the system needs a lower latency. It shows a lower RTT than the other stateful and stateless methods when the LB is not the bottleneck of the system.

The project was demanding, connecting many concepts and working with broad code-bases as FastClick or DPDK is not easy. However, our group managed to implement the methods we were

required to. We have unfortunately not been able to obtain conclusive results for Cuckoo++ and HW classification, not being able to perform a general analysis. Nevertheless the results obtained will be helpful to guide future work towards the improvement of LBs performance.

If we could do this project again we would certainly start the implementation of new methods sooner. Understanding the code took time, we needed an iterative process to overcome the learning errors. Starting earlier with the implementation would have helped to understand concepts of how LBs work. We would also suggest to use another approach for testing as TRex may not be the best tool to analyze LBs as we are going to explain in the limitations.

Overall, we found out that different load balancing methods can influence significantly on the performance. Improving or changing the methods may have a positive impact on the LB, giving more throughput and less latency to the system.

6.2 Limitations

Since we used TRex as traffic generator for our tests, we encountered different problems when doing the measurements. The main issue is that our tests did not track the sources and destinations of the packets, meaning that our results did not take into account if each source corresponds to each destination. Another drawback of using TRex is the instability of the latency data. We changed how TRex save the latency results to reduce the variance and get more consistency but, even after these modifications, some results still have a lot of variance. We believe that these instability is caused by the mechanism used by TRex to measure latency and, thus, a further investigation in this area may be done.

Finally, when 8 cores are used, the stateful LBs with DPDK's *MULTI_WRITER_ADD + RW_CONCURRENCY_LF* flags and the *Per Core Duplication* reach our testbed throughput limit. Hitting this limit means that we have not been able to analyse different throughput behaviours between these methods when running with 8 cores.

6.3 Future Work

6.3.1 Hardware classification implementation

While developing the hardware classification LB we encountered many problems, as a result we have only been able to develop a LB that has a very low throughput (less than 1 Gbps) compared to the other methods. This method should have been able to provide a higher throughput and a lower latency than the other stateful methods. Regarding this point, it would be interesting to investigate and develop this method in order to improve our results.

6.3.2 TRex measurement mechanisms

After having analysed the latency data of our tests, we found a significant instability. After some research and tests we changed how the latency data was saved to add precision, but it was not enough as some measurements still have high latency variance.

Since TRex is also an open source project, a future work could be done in the implementation of another latency measurement system both for stateful and stateless mode, with the aim of having a better and reliable traffic generator.

6.3.3 Flow ageing implementation

During the evaluation of the performances in terms of throughput and latency, we noticed an instability. In Figure 6.1 the throughput is reported, to show the trend over time.

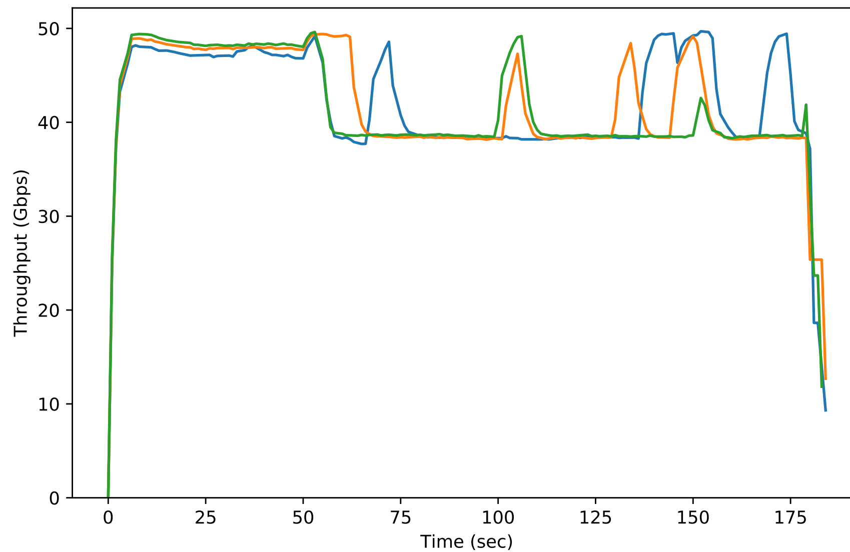


Figure 6.1: Throughput over time of the same test ran 3 times

We investigated this behavior through profiling and we reported the screenshot of the perf command in Figure 6.2.

The conclusion that we got from the profiling is that at some point the flow table starts to be full, and a lot of time is spent in operations such as `std::_Function_handler` that is responsible to check for old flow in the flow table.

From our test and also from Figure 6.1 we noticed that this behavior always started approximately after sixty seconds of execution, that is the timeout set by default for the *TimerWheel*.

(a) Profiling before the unstable part

(b) Profiling during the unstable part

Figure 6.2: Profiling of LB with Per Core Duplication

The aim of the timer wheel is to free the position in the flow table for the expired flows. After a discussion with the author of the code, we discovered that it was a new components and that it has not been tested and optimized yet. Future work could be done around this element, analysing the performances and the behavior of various flow timeout techniques, to obtain more accurate data about how a LB—based on FastClick—could work in a real environment.

6.4 Reflections

We aimed to help with the improvement of software LBs performance. By doing so we wanted to guarantee that the requests sent to the applications are answered in the most efficient way and in the shortest possible time. We also intended to support as many clients as possible, ensuring a constant and high throughput. First because it would increase the overall experience of the clients, secondly because it could build loyalty towards the websites or applications that the clients are using.

Trying to improve the performance of software LBs is economic and sustainable. Compared to their physical counterpart, where improvement can be done through deployment of numerous physical LBs, the malleability and flexibility of the software can provide easier testing of the implemented methods that are nearly ready-to-use. In this way the cost of buying multiple physical devices would be saved and the carbon footprint would be lowered as well.

Concerning the ethical aspect, all the implementations of the methods are open source. This means that anybody can contribute to the improvement of the project. All the results and scripts used to run the tests are shared, easily allowing other people to replicate our experiments and ensuring that no information has been hidden to the public.

Bibliography

[1] Sukrati Jain and Ashendra K Saxena. "A Survey of Load Balancing Challenges in Cloud Environment". In: *2016 International Conference System Modeling & Advancement in Research Trends (SMART)* (2016).

[2] D.E. Eisenbud et al. "Maglev: A Fast and Reliable Software Network Load Balancer". In: *NSDI* (2016).

[3] Parveen Patel et al. "Ananta: Cloud Scale Load Balancing". In: *ACM SIGCOMM Computer Communication Review* 43.4 (2013), pp. 207–218.

[4] *Load Balancer Pricing*. URL: `https://kemptechnologies.com/`.

[5] Tom Barbette, Cyril Soldani, and Laurent Mathy. "Fast Userspace Packet Processing". In: *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)* (2015).

[6] Eddie Kohler et al. "The Click Modular Router". In: *ACM Transactions on Computer Systems* 18.3 (2000), pp. 263–297.

[7] *Intel Data Plane Development Kit*. URL: `https://www.dpdk.org/`.

[8] Christoph Lameter. "NUMA (Non-Uniform Memory Access): An Overview". In: *Queue* 11.7 (2013), p. 40.

[9] Hyogi Jung. "Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update 2017–2022 White Paper". In: (). URL: `https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-738429.html`.

[10] Shanti Swaroop Moharana, Rajadeepan D Ramesh, and Digamber Powar. "Analysis of Load Balancers in Cloud Computing". In: *International Journal of Computer Science and Engineering 2.2* (2013).

[11] *Load Balancer Definition*. URL: `https://www.nginx.com/resources/glossary/load-balancing/`.

[12] Jun Xu and M Singhal. "Cost-Effective Flow Table Designs for High-Speed Routers: Architecture and Performance Evaluation". In: *IEEE Transactions on Computers* 37.9 (2002), pp. 1089–1099.

[13] Pankaj And Gupta. “Packet Classification Using Hierarchical Intelligent Cuttings”. In: *Hot Interconnects VII* (1999).

[14] Sumeet Singh et al. “Packet Classification Using Multidimensional Cutting”. In: *SIGCOMM* (2003).

[15] F Baboescu, Sumeet Singh, and G Varghese. “Packet Classification for Core Routers: Is There an Alternative to CAMs?” In: *IEEE INFOCOM* 1 (2003), pp. 53–63.

[16] K LeFevre, D J DeWitt, and R Ramakrishnan. “Mondrian Multidimensional K-Anonymity”. In: *International Conference on Data Engineering (ICDE)* 6 (2006).

[17] Zhi Liu et al. “BitCuts: A Fast Packet Classification Algorithm Using Bit-Level Cutting”. In: *Computer Communications* 109 (2017), pp. 38–52.

[18] Rasmus Pagh and Flemming Friche Rodler. “Cuckoo Hashing”. In: *Journal of Algorithms* 51.2 (2004), pp. 122–144.

[19] Salvatore Pontarelli, Pedro Reviriego, and Juan Antonio Maestro. “Parallel D-Pipeline: A Cuckoo Hashing Implementation for Increased Throughput”. In: *IEEE Transactions on Computers* 65.1 (2016), pp. 326–331.

[20] Bin Fan, David G. Andersen, and Michael Kaminsky. “MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing”. In: *NSDI* (2013).

[21] Brad And Fitzpatrick. “Memcached. A Distributed Memory Object Caching System”. In: (2011).

[22] Sorav Bansal and Dharmendra Modha. “CAR: Clock with Adaptive Replacement”. In: *Proceedings of the 3rd USENIX Conference on File and Storage Technologies* (2004).

[23] Dong Zhou et al. “Scalable, High Performance Ethernet Forwarding with CuckooSwitch”. In: *CoNEXT* (2013).

[24] Nicolas Le Scouarnec. “Cuckoo++ Hash Tables: High-Performance Hash Tables for Networking Applications”. In: *CoRR* (2017).

[25] “iPerf - the TCP, UDP and SCTP Network Bandwidth Measurement Tool”. In: (). URL: `https://iperf.fr/`.

[26] *TRex - Realistic Traffic Generator, from Cisco*. URL: `https://trex-tgn.cisco.com/`.

[27] James Scotland. “Exploring the Philosophical Underpinnings of Research: Relating Ontology and Epistemology to the Methodology and Methods of the Scientific, Interpretive, and Critical Research Paradigms”. In: *English Language Teaching* 5.9 (2012).

[28] *Linux Network Namespaces*. URL: `http://man7.org/linux/man-pages/man8/ip-netns.8.html`.

[29] *Intel® Xeon® Processor E5-2667 v3 (20 MB Cache, 1.2 Ghz - 3.2 Ghz)*. URL: `https://ark.intel.com/products/83361/`.

[30] *Mellanox Technologies MT27700 Family [ConnectX®-4]*. URL: `https://www.mellanox.com/page/products_dyn?product_family=204`.

[31] *Mellanox Technologies MT27800 Family [ConnectX®-5].* URL: https://www.mellanox.com/page/products_dyn?product_family=260.

[32] *Forked TRex Repository.* URL: https://github.com/MassimoGirondi/trex-core/tree/linear_histogram.

[33] *Perf Linux Tool.* URL: https://perf.wiki.kernel.org/index.php/Main_Page.

[34] *FastClick Elements Description.* URL: https://github.com/kohler/click/wiki.

[35] Greg Gagne Abraham Silberschatz Peter B. Galvin. *Operating System Concepts, 9th Edition.* WILEY, 2013.

[36] *DPDK Cuckoo Multithread Support.* URL: https://doc.dpdk.org/guides/prog_guide/hash_lib.html#multi-thread-support.

[37] *Introduction to Receive Side Scaling.* URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling.

[38] *DPDK Actions.* URL: https://doc.dpdk.org/guides/prog_guide/rte_flow.html#action-set-ipv4-src.

[39] *Elastic Flow Distributor Library.* URL: https://doc.dpdk.org/guides/prog_guide/efd_lib.html.

[40] *Load Balancer Sample Application.* URL: https://doc.dpdk.org/guides-18.02/sample_app_ug/load_balancer.html.

[41] *DPDK supported hardware offloads on Mellanox interfaces.* URL: https://doc.dpdk.org/guides/nics/mlx5.html#supported-hardware-offloads.

[42] Alex D. Breslow et al. "Horton Tables: Fast Hash Tables for in-Memory Data-Intensive Computing". In: *Proc. USENIX Annu. Tech. Conf.* (2016).

[43] Xiaozhou Li et al. "Algorithmic Improvements for Fast Concurrent Cuckoo Hashing". In: *Proceedings of the Ninth European Conference on Computer Systems - EuroSys '14* (2014).

[44] Rina Panigrahy. "Efficient Hashing with Lookups in Two Memory Accesses". In: (2004).

[45] *Libcuckoo.* URL: https://github.com/efficient/libcuckoo.

Glossary

5-tuple A tuple consisted by source and destination IPs, socket ports and protocol type that can be used to identify a single flow passing through a network.

flow A stream of data that needs to be treated coherently.

profiling The analysis of the CPU usage by the different functions of a program.

Acronyms

CDF Cumulative Distribution Function.

CPU Control Process Unit.

DB Database.

DPDK Intel. Data Plane Development Kit.

ICT Information and Communication Technology.

LB Load Balancer.

NIC Network Interface Controller.

NUMA Non-Uniform Memory Access.

RSS Receive Side Scaling.

RTT Round Trip Time.

StF LF Stateful LB with `RW_CONCURRENCY_LF` flag set.

StF MW Stateful LB with `MULTI_WRITER_ADD` flag set.

StF MW LF Stateful LB with `RW_CONCURRENCY_LF` and `MULTI_WRITER_ADD` flag set.

StF MW RWC Stateful LB with `RW_CONCURRENCY` and `MULTI_WRITER_ADD` flag set.

StF NF Stateful LB (No Flag).

StF RWC Stateful LB with `RW_CONCURRENCY` flag set.

TTL Time To Live.