A comparison of multi-core flow classification methods for load balancing

Max Crone <mccrone@kth.se>
Anna Fernandez-Rajal <annafris@kth.se>
Quang Le <dqle@kth.se>
Felix Maurer <fkjma@kth.se>
Zhangchi Qin <zhangchi@kth.se>
Kibria Shuvo <gkshuvo@kth.se>

Communication System Design – Final Report

IK2200 Communication System Design
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden

7 January 2021

Academic supervisor: Tom Barbette

Abstract

Load balancers enable a high number of parallel requests to a web application by distributing the requests to multiple backend servers. Stateful load balancers keep track of the selected server for a request in the flow table. As the flow table is accessed for each packet, its implementation is crucial for the performance of the load balancer. In this report, we evaluate and compare three single-core implementations of flow tables in a load balancer, based on C++ unordered maps, Cuckoo hash maps, and Hopscotch hash maps. Later, we scale the load balancer to multiple cores using locks or separate hash maps per running core. The different scaling mechanisms are evaluated and compared in terms of throughput and latency. We also include an implementation based on Cuckoo++ hash maps in our comparisons, which we selected after reviewing the current literature. All implementations use FastClick and are evaluated with traffic generated by TRex. The evaluation combines results from traffic measurement with results from profiling the various implementations. We show that the implementation of a flow table based on Hopscotch hash maps performs best, while Cuckoo++ was unexpectedly shown to perform worse than Cuckoo. For the scaling mechanisms, we show that one flow table per-core outperforms the locking based approach, because it allows all cores to process packets independently from each other.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Internet traffic has been increasing over the last years, and is expected to grow even further in the coming years [9]. More people are becoming connected to the Internet. Machine-to-machine traffic is expected to increase significantly as a result of the consumer and the industrial Internet of Things. Multi-media streaming is growing in bandwidth needs. The world is far beyond a point where single servers can accommodate the Internet applications of our time. Large data centers are being built all around the world to house rows of computer hardware. The purpose of these servers is to handle the large amount of Internet traffic.

In order to effectively leverage data centers to handle these large amounts of traffic, network packets need to be distributed among all individual servers. Load balancers are a critical part in the network that facilitate this. Their purpose is to prevent individual servers from being overloaded with requests, by distributing the traffic over many servers. In many cases, the traffic distribution needs to be stored for correct handling and continuity of a network connection in future communication. These records are stored in flow tables on the load balancer. The implementational details of a flow table will contribute to the efficiency of a load balancer. Therefore, the choice of flow table implementation becomes important.

In this report, we implement multiple flow tables and compare them on a testbed with network traffic going up to 100 Gbps. Subsequently, we implement and compare different methods of scaling the flow tables' operation across multiple CPU cores. The results of these comparisons will contribute to the use of more efficient flow tables, and thus to faster and more sustainable operation of load balancers.

1.1 Problem statement

Despite the existence of multiple traffic classification methods for load balancers, there is a limited number of studies to compare these techniques to understand which load-balancing method is the most efficient for each load-balancing scenario. Therefore, it is necessary to conduct more research that compares different load-balancing methods.

1.2 Goals

There are three main goals for this project.

Firstly, we aim to study, implement and compare multiple flow tables on a physical testbed. We use the FastClick [6] framework for routing network packets through the load balancer and implementing the flow tables.

Secondly, we implement and evaluate multiple methods of scaling the flow tables to leverage multi-core processors.

Lastly, we perform a literature study into state-of-the-art flow table data structures, implementing one promising option and comparing its performance directly with other flow tables.

1.3 Methodology

This project is set up as quantitative research. All flow classification methods will be compared based on a set of metrics obtained from the experiments we conducted. In these experiments, we will vary the underlying parameters in order to observe the change in performance for all solutions. The results serve as the foundation on which we base our final conclusions and recommendations.

1.4 Limitations

All flow tables are implemented as part of the FastClick framework, which uses DPDK for packet processing. This work does not compare FastClick with alternatives, as it is beyond the scope of this project.

Additionally, we do not evaluate different load balancing approaches. The distribution of network traffic in our system does not take into account any contextual parameters besides the identifying five-tuple of network packets. This means we do not consider CPU load or other indicators of actual load. Neither do we use

or evaluate round-robin approaches. The focus of this project is on the flow table storage, and less on the different solutions for distribution mechanism.

Finally, although we briefly discuss other methods in Section 2.4, this work only compares solutions based on hash tables.

1.5 Sustainability and ethical considerations

Data centers have become essential computing infrastructure that enable the IT industry. As the sector grows, these data centers also increase in scale. Data centers thus require an increasingly large energy budget to operate. A data center's energy budget is made up of important components that include cooling, servers and storage, and networking hardware [12]. Efficient use of these resources, means that a data center can operate more energy-efficiently. This project contributes to more efficient use of the available servers and networking hardware. Our work enables organizations that operate at large scale to choose the more efficient flow classification methods for their load balancers, which will increase their overall energy-efficiency. This means that an equal amount of work can be done while using less electricity, thus contributing to a more sustainable world.

All measurements performed as part of this project are conducted on a dedicated testbed. Thus, none of these measurements negatively impacts any other processes. Nor does the project have access to any personal or other privacy-sensitive data. Finally, none of the synthetic data used during measurements contains any amount of personal or privacy-sensitive data.

This work extends upon the projects Click and FastClick, which are licensed under an amended version of the MIT license. It allows "to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so". The license will be provided along with the publicly distributed source code of this project.

1.6 Structure of the report

The remainder of this report is organized as follows. Chapter 2 provides technical background on load balancers, DPDK, and FastClick. It additionally presents related scientific work. Chapter 3 describes the experimental setup, including a discussion on traffic generation, and our evaluative approach. Chapter 4 compares different proposed hash tables for implementing flow tables and selects one of them

to include in our evaluation. Later details the implementation of our single-core and multi-core load balancer flow tables.

In Chapter 5, we evaluate the performance of all implementations and compare the results. Chapter 6 concludes the comparison and provides recommendations.

Chapter 2

Background

2.1 Load balancers

Load balancers (LBs) are devices responsible for distributing incoming network requests over several servers [11]. They allow a distributed system to handle traffic at a greater scale by ensuring that no server is overloaded leading to its unavailability. They aim to improve the performance of the different applications by managing the traffic on the network and making it possible for complex environments to work smoothly.

Load balancers fall into two categories regarding how they distribute the workflow: stateless and stateful. While stateless load balancers do not ensure an equal load distribution, Section 2.1.1, the selection of servers used on stateful load balancers can achieve a more even workload distribution since it is also based on load metrics, Section 2.1.2.

By continuously routing the incoming request to the different available servers, load balancers prevent overworking a specific server and possible bottlenecks on the network in order to optimize the network's performance. As previously mentioned, depending on whether the LB keeps track of the sessions information, we can consider the following types of load balancers.

2.1.1 Stateless load balancers

Stateless load balancers are simpler, faster, and have a lower computational cost than stateful ones, but the load distribution can be less accurate depending on the situation by treating the requests equally. They send the request of a certain user to the same server, by creating a hash of the user based on the protocol in use,

in general TCP or UDP; the IP addresses; and the ports of both the source and destination. Based on this hash function there is no certainty that the distribution among the users will be done uniformly. This can lead to an unequal load distribution if the users do not have the same incoming traffic on the network, and so can overload some of the servers while leaving others relatively idle [45]. On the upper hand, they do not require retention of the session information, which saves memory. Figure 2.1 illustrates an example of a stateless load balancer. After receiving a packet from data, the load balancer retrieve several fields in the IP header of the packet and apply a hashing algorithm on these extracted values to determine the receiving server. As a result, the packets are distributed across different servers regardless of the current state of each server (e.g., traffic load).

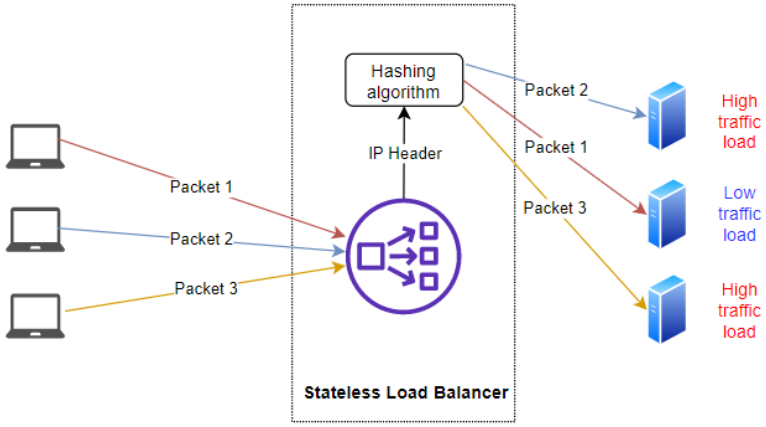


Figure 2.1: Stateless load balancer

2.1.2 Stateful load balancers

A stateful load balancer keeps track of each session, routing all packets from one client to the same machine. The machine will remember the state of that client [33]. These ones can pick different servers by, for example, looking into the current workload of each server, or into their previous request in order to manage them in a better way. New packets are assigned to new servers while the ones that already have an existing connection on the flow table are sent to the already established server. Thus when a packet arrives at the load balancer, it firstly checks whether the flow is new or already has a server assigned.

However, by giving better results on distributing the flows on the network, they require more computational resources and memory. Stateful load balancers need to calculate the hash of each of the incoming requests and then check if it is a new

entry or not, and send it to the assigned server. Each of the searches on the flow table can lead to a high cost depending on the number of entries.

Figure 2.2 illustrates an example of a stateful load balancer. Upon receiving a packet, the load balancer utilizes several fields of the IP header, a hashing algorithm, and other metrics (e.g., server load) as the inputs to determine the representation of the packets in the flow table. Incorporating external metrics as an input allows for effective traffic distribution where all three new flows are sent to the server with the lowest load.

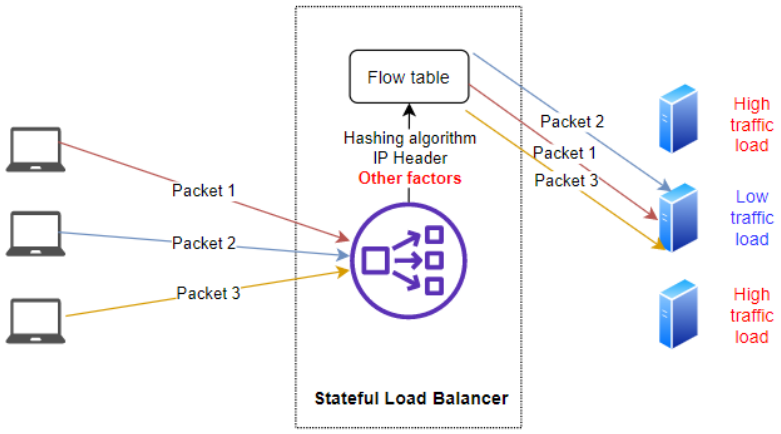


Figure 2.2: Stateful load balancer

Flow tables are commonly implemented using hash tables. Hash tables are data structures to store a mapping from keys to respective values. They usually provide efficient algorithms for the lookup of entries. Many different types of hash tables exist that differ in the storage of the entries and therefore in the algorithms to access them.

All hash tables store their data in a number of so called buckets. The implementation of the buckets varies between the different types of hash tables. In general, a single bucket should only contain a few elements to maintain the efficiency of the lookup operation. The bucket that an entry should be stored in is determined by a hash function. The hash function is the main reason for the efficiency of hash tables because it is only computed once regardless of the size of the table. This function assigns a number to each key which is then used to select the target bucket. A good hash function achieves a uniform distribution of the entries over the buckets. However, even a good hash function can not avoid that multiple entries are mapped to the same bucket. Such situations are called collisions. Most hash tables include a mechanism to handle collisions and store all the colliding entries. The collision

resolution algorithms often have a severe impact on the performance when the number of entries approaches the number of buckets.

The flow table of a load balancer uses the identifying five-tuple of a flow (i.e., source IP address, source port, destination IP address, destination port, and protocol) as the key and stores the selected backend server and possibly other state information as the value. For every arriving packet, the load balancer needs to perform a lookup on the flow table to check if the flow was seen before and a decision on the backend server was already made. Thus, lookups need to be as fast as possible even with many entries in the table. The second important operation is the insertion of new flows into the flow table. This also needs to happen efficiently considering the most common deployment of load balancers is to distribute the connections from a huge number of clients to several backend servers. Therefore, special attention needs to be payed to the efficiency of the lookup and insert operations when choosing a hash table to implement a flow table.

2.2 Data Plane Development Kit

This project uses Data Plane Development Kit (DPDK) to allow high-speed packet processing. In the x86 structure, the traditional way of processing data packets is the CPU interrupt mode, i.e., the network card driver receives the data packet and informs the CPU to process through an interrupt, and then the CPU copies the data and delivers it to the protocol stack. However, once the amount of data becomes huge, this method will generate a large number of CPU interrupts, making the CPU unable to run other programs [13].

DPDK uses the polling method for implementing the data packet processing. By reloading the network card driver, it will not interrupt the CPU after receiving the data packet, but stores the data packet into the memory by zero-copy technique [50], which maps buffer data to user space straightforwardly without copying in the kernel space. Meanwhile, the user space program, i.e. FastClick, can read data packets directly from the memory through the Network Interface Controller (NIC). Therefore, the stack does not need to receive the packets, and packet counters increase will not be shown in ifconfig. This processing method saves CPU interrupt and memory copy time. Moreover, DPDK provides a simple and efficient data packet processing method for the application layer, making the development of network applications more convenient.

2.3 FastClick

FastClick is a library based on Click, a packet processor offering a simpler configuration, integrating Click with Netmap and DPDK [6]. It automatically handles multi-queuing, core affinity, and batching while keeping the compatibility with Click. Its integration with DPDK allows it to retrieve and deliver the packets on its own, without having to do a system call after each of the packet processing. We are going to use FastClick to implement different load balancers as detailed in Chapter 5, to scale from single-core to multi-core and taking advantage of its elements structure to perform different network functions.

Click is a software architecture for creating routers, based on packet processing modules called elements [32]. In order to configure a Click router, there is a choice of elements and a connection between them, calling the ends connections and representing paths for the packet delivery. This flexible and modular software allows users to create completely new elements or based on the existing ones by piping them together. Each of the Click elements represents a unit of router processing, and each action is encapsulated in one.

2.4 Related work

Multiple methods have been developed for flow classification nowadays. Their approaches differ, which leads to different balances of performance metrics such as execution time and memory usage. Methods can be distinguished as software-based or hardware-based, which are discussed separately in the following sections.

2.4.1 Hardware-based flow classification

Conventional hardware-based load balancers [1, 38] mainly make use of Application-Specific Integrated Circuit (ASIC) and Ternary Content Addressable Memory (TCAM). In general, although hardware-based classifiers can achieve high speeds and throughput rates up to 100 MPPS, they cannot be easily developed and customized given the limited resources on the chip. Moreover, these systems carry high costs, and have limitations on scalability, availability, flexibility, and cost-efficiency [16]. Hence, software-based methods have increased in popularity in recent years.

2.4.2 Software based flow classification

Compared with hardware-based flow classification, software-based classifiers have more extensibility. However, they do not perform as efficiently in networks with

high bandwidth due to the low speed of the serial processing of instructions in CPUs. Accelerating the software-based classifiers, therefore, has resulted in considerable research with the aim of developing methods to increase the speed of its algorithms. We can roughly divide them into two approaches; decision-trees and hash table data structures.

2.4.2.1 Decision trees

Decision tree-based algorithms are considered an important class of software-based classification methods. In this type of classification, the rule sets are stored in the search tree based on binary patterns in the rule fields. To find the rule that best matches the incoming packet, the tree is traversed based on the binary content of the fields in question [35]. Researcher have developed various tree-based algorithms such as AQT [36], HiCuts [30], Hyper-Cuts [46], and Bitcuts [37]. These algorithms manage to obtain efficient search methods by first using the geometric representation of rules, and then constructing the corresponding decision tree. However, existing decision-tree algorithms still have inefficiencies in their space decomposition schemes, resulting in redundant cutting or degraded classification speed. Hence, the insertion time for flows costs more, which is extremely relevant for our use case of flow classification.

2.4.2.2 Hash tables

Most existing load balancing algorithms use hash tables to store connection states as the data plane solution [44]. In large-scale datacenters, there are tens of thousands of servers and hundreds of load balancers. These stateful load balancers may experience large memory cost of storing flow states or low capacity of flow processing. To solve that, some load balancers use digests of connections rather than full connection state, i.e. five-tuples, to reduce memory cost and improve throughput, e.g. Google [16] and Microsoft [19]. However, this may result in violations of flow consistency due to digest collisions. On the other hand, it will cause frequent data plane updates due to large numbers of new connections.

Concury [44] adopts Othello Hashing [18], which makes use of Bloomier filters, to alleviate this problem. It is deployed for the large number of network states with frequent connection arrivals, which performs consistently under infrequent network updates. Also, the typical data structure that can be used to maintain states in the above methods is Cuckoo hashing. Cuckoo hashing is an open-addressed hashing technique with high memory efficiency and $O(1)$ amortized insertion time [34], which represents the average level of multiple insertions. Hash tables based on open addressing are those that store all the elements within the table. This is a collision handling technique where no element is stored outside the limits of the table, and

thus the memory usage can be predicted. In the case of collisions, there are different approaches to compute new positions (e.g Cuckoo and Hopscotch). Multiple improvements and extensions based on Cuckoo hashing have been developed to improve performance in practice and for higher load factors. Examples include MemC3 [18], CUCKOOSWITCH [52], and Cuckoo++ [42],

Chapter 3

Experimental setup

In order to directly compare the flow classification mechanisms in this report, we collect measurements of the respective execution of FastClick with all of the mechanisms on a testbed as described in Section 3.1. These measurements come from two distinct sources.

Firstly, we gather statistics from our traffic generation software that sum up throughput and latency metrics. This measures the effectiveness of FastClick and the respective flow classification mechanism; how much traffic it was able to handle and at which speed it did so. We will repeat the measurements over multiple iterations in order to build confidence in the results. Section 3.3 describes this measurement setup.

Secondly, we profile FastClick and the respective flow classification mechanisms during execution using perf [40]. This allows us to characterize the functions being called while the load balancer is operational, which will provide insight into the CPU occupation of FastClick. These measurements will be used to compare the efficiency of different flow classification mechanisms. Additionally, profiling using perf allows us to determine to which extent FastClick is the limiting factor for the traffic flow through the load balancer. Section 3.4 provides more details on the profiling process.

3.1 Testbed

The setup for measuring the performance of the different flow table implementations consists of two physical hosts, as we can see in Figure 3.1. Both machines have one Intel Xeon Gold 5217 processor and 6 x 8 GB memory. They are running

Ubuntu Linux 18.04.1 with kernel version 5.4.0. One host serves as a traffic generator producing the network traffic needed for the evaluation of the implementations. The other host is the device under test running the load balancer implementations in FastClick.

Each host is equipped with a Mellanox ConnectX-5 network interface card (NIC) with two 100 Gbit/s QSFP28 ports. The hosts are connected with two directly attached cables using these NICs. The structure of the testbed is shown in Figure 3.1.

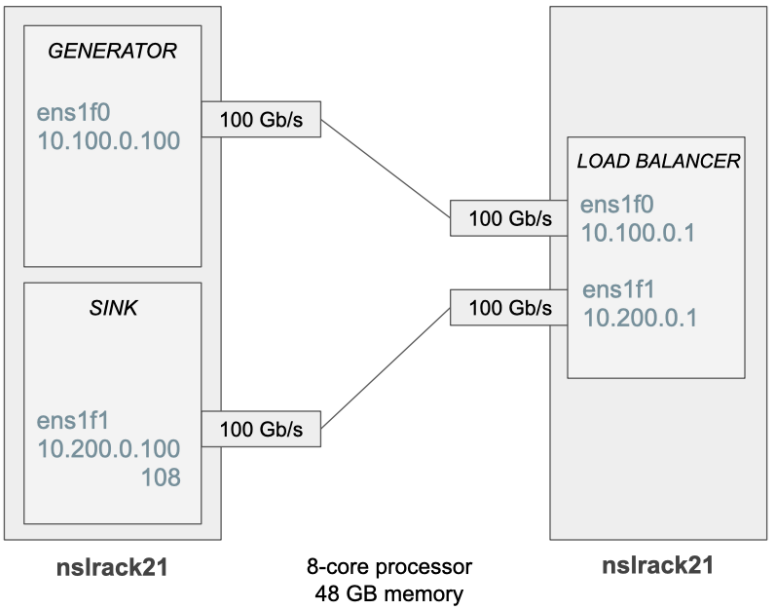


Figure 3.1: Setup of the testbed

3.2 FastClick configuration

As mentioned in earlier chapter, we use FastClick to emulate the load balancer. The FastClick instance is built on version 1.0.6 with the support for DPDK (version 20.08). FastClick is executed with enabled DPDK support, eight memory lanes, two network interfaces, and the number of CPU cores depends on the testing scenario.

Our FastClick configuration is developed based on the modular configuration technique which includes: (1) common Click configuration files for code reuse, and (2) one separate configuration file for each load balancing scenario.

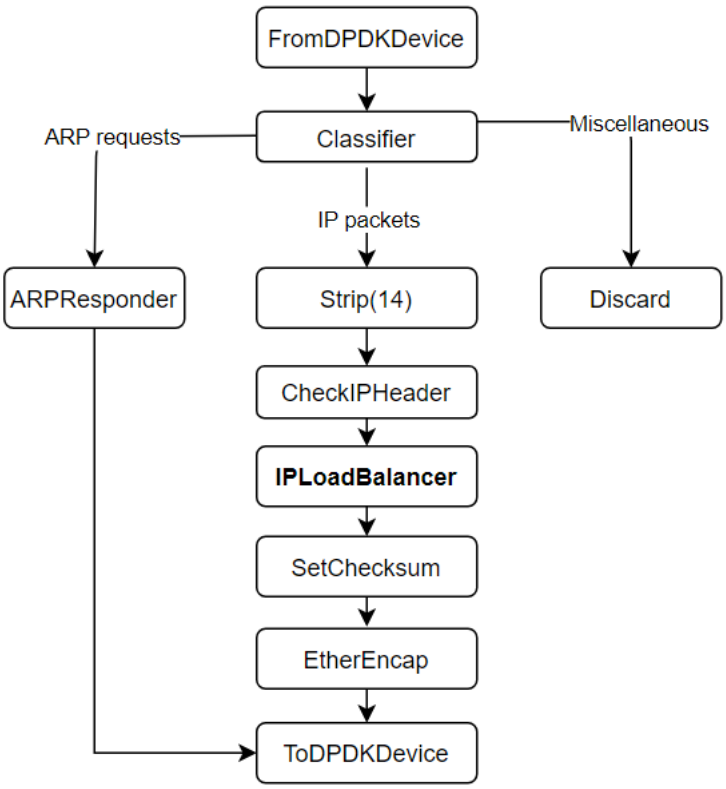


Figure 3.2: Processing flow of stateless load balancer configuration

3.2.1 Stateless load balancer configuration

The stateless load balancer is implemented using the built-in FastClick `IPLoadBalancer` element [5]. Arguments for the `IPLoadBalancer` element include the virtual IP address of the load balancer (i.e, the IP that the generator sends packets to), and the list of IP addresses of the servers to receive traffic (i.e, the sinks in our topology).

The process flow of FastClick is described in Figure 3.2. After receiving a packet from the network interface, FastClick validates IP Header field of the packet. After that, the packet is passed by the `IPLoadBalancer` element to rewrite the destination IP Address field of the packet to a server based on the hash table. Following this, the checksum of the packet is recalculated before the MAC addresses are modified by `EtherEncap` to be sent out of the interface. The reverse direction follows the same processing flow of the forward direction, with the use of `IPLoadBalancerReverse` to rewrite the source IP address of the server to the virtual IP of the load balancer. In this implementation, we use `EtherEncap` with static MAC addresses to bypass the ARP resolution process, which include

considerable CPU usage during FastClick execution, to analyze the main functions of FastClick. Regarding checksum calculation, the SetChecksum function would invoke `ResetIPChecksum` to offload checksum calculation to the hardware instead of using the CPU to calculate it.

3.2.2 Stateful load balancer configuration

The hashtable load balancer is implemented using FastClick `FlowIPManager` and `FlowIPLoadBalancer` [3] elements. `FlowIPManager` is the element that manages the flow table. Checks whether a packet belongs to an existing flow, and adds a new entry if it belongs to a new one. The assigned flow is used on the `FlowIPLoadBalancer` to do the load balancing and so set its destination address to the packet. Both these elements were added with the integration of MiddleClick [7] into FastClick.

The processing flow of this load balancer is similar to the stateless load balancer in the previous section, aside from the `FlowIPLoadBalancer` element replacing `IPLoadBalancer` to perform the load-balancing task. In the reverse direction of the packet, `FlowIPLoadBalancerReverse` [4] is utilized to replace the source IP address of the server back to the virtual IP of the load balancer. The detailed processing flow is described in Figure 3.3.

3.3 Traffic generation

The network traffic is generated with TRex [10]. This is an open source traffic generation software developed by Cisco. It makes use of DPDK to generate stateless and stateful network traffic at high rates. In stateless mode, TRex generates configured packets without taking any context into account. In contrast, the stateful mode allows to generate more complex traffic patterns that replicate stateful network protocols. It allows, for example, to send packets following a more strict timing or answer incoming packets with reply packets.

For evaluating the performance of different flow table implementations in a load balancer, it is not necessary to generate stateful network traffic. Instead, stateless traffic can be used, which consumes fewer resources on the traffic generator to reach the same data rates. This is possible because packets flowing through the load balancer in one direction in a particular order are sufficient to trigger the interesting flow table operations, i.e., inserts and lookups. The stateful mode would allow to send replies through the load balancer. But such replies would not trigger additional operations on the flow table and are thus not necessary.

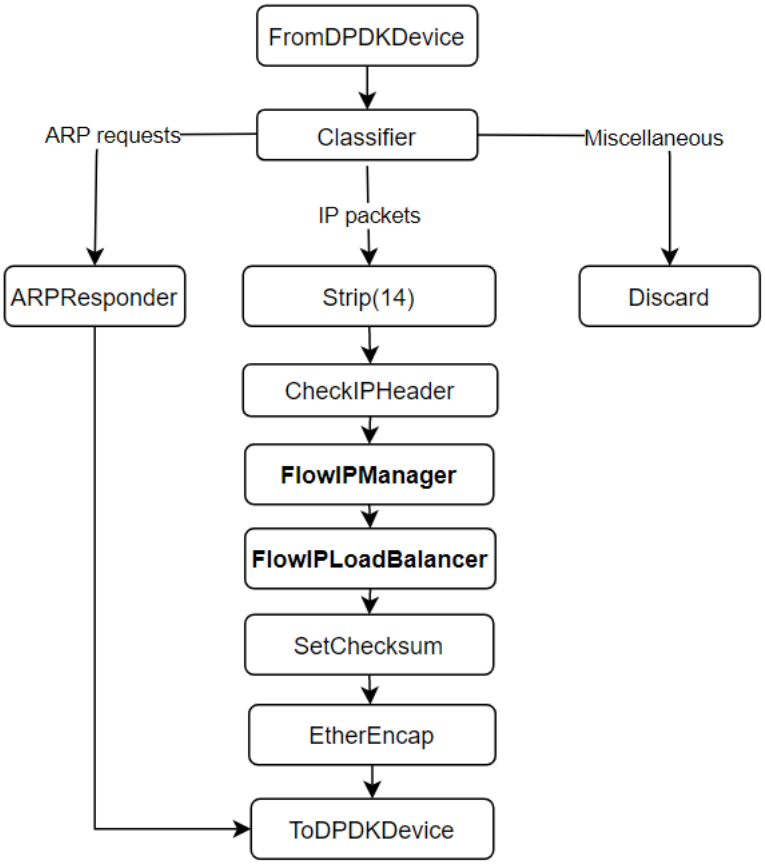


Figure 3.3: Processing flow of stateful load balancer configuration

TRex uses a client-server architecture for generating traffic. The server is implemented in C++ and performs the high-speed generation of packets. The client configures the packets to be sent and controls the packet generation. It can be implemented in Python using an API that is provided by TRex.

3.3.1 Stateless traffic generation with TRex

The stateless traffic generation mode of TRex offers sufficient configuration options with minimal resource consumption.

The configuration of stateless traffic that TRex generates happens by attaching streams of packets to ports. Ports represent, as with DPDK, the network interfaces that will be used. A stream has one template for the packets it generates and a mode for sending them. The mode can be for example continuous, single burst, or multi burst. Streams can also have a set of modifications that the field engine applies to the packets and a mode of stream statistics collection.

TRex can either collect no statistics on a stream, basic statistics like packet and data rate in hardware, or more detailed latency statistics in software. The number of streams that basic statistics can be collected on are limited by the NICs. The Mellanox ConnectX-5 in our setup can handle this for up to 127 streams. As the basic statistics are collected in hardware, the packet rate of the streams does not matter. Collecting latency statistics also activates the collection of basic statistics which makes them also count towards the limitations of the NICs. In contrast to the basic statistics however, the streams with latency statistics should also be limited in terms of packet rate, because each packet of the stream needs to be received and handled by a software thread. The TRex documentation recommends not to exceed five million packets per second combined for all streams with latency statistics [51].

Multiple streams can be attached to the same port. The streams can be executed in parallel, but the completion of a stream can also trigger the start of another stream. This way, streams can be configured that are dependent on each other.

For testing the flow table implementations, we required a testing application that allows to generate traffic with different packet sizes, produce a configurable number of flows, collect as detailed statistical data as possible, and can reach data rates up to 100 Gbit/s. The following sections describe the streams that we use to achieve the requirements and their integration in our testing application.

3.3.1.1 TRex stateless stream configuration

The performance of a network device can heavily depend on the size of the packets it handles. Therefore, the packet size that a stream generates can be adjusted to the test case by setting a parameter when the stream is created.

Especially with small packets at high data rates, we would exceed the maximum packet rate for the latency statistics. Therefore, our implementation uses two streams: one traffic stream and one measurement stream. Both streams generate the same packets but at different rates. The traffic stream generates packets at very high rates to reach the aspired data rates while only collecting basic statistics. In contrast, the measurement stream generates packets at a fixed rate of 10 000 packets per second while collecting detailed latency statistics. This means that the latency introduced by the load balancer is only measured every 100 µs. The latency statistics collected from a small subset are only representative for all packets if they are handled equally by the load balancer. For this assumption to hold, the packets generated by the two streams should be as similar as possible. Therefore, we do not base any substantial modifications of the packets on the type of the stream.

However, we need to make sure that the different streams do not influence each others packet handling. This could, for example, happen when a packet of a measurement stream belongs to the same flow as earlier packets from a traffic stream. To ensure that the streams have disjoint sets of flows, they use source addresses from different IP ranges.

The measurement streams have an initial delay of ten seconds before the first packet is sent. This allows the traffic stream to reach its desired rate and stabilize its operation before we start to collect the statistical data for our analysis.

3.3.1.2 Performance issues with the stateless mode

The described configuration shows issues in tests with high data rates as they are expected for the multi-core implementations. When the data rate of the generated traffic exceeded 32 Gbps, the measurements show a high variance and a decreasing throughput with increasing data rates. In this setup, TRex is capable of generating traffic at the high data rate and the load balancer is able to forward data rates above this threshold. In addition, when exceeding the data rate limits of the load balancer, it shows a steady throughput at its maximum capacity with low variance. Therefore, the issues are most probably a result of TRex not being able to receive traffic at high rates in stateless mode.

We describe our testing setup based on the stateful mode of TRex in the following sections.

3.3.2 Stateful traffic generation with TRex

The core idea of the stateful traffic generation mode is to replay existing traffic captures. The captures need to be present in the PCAP-format before starting the traffic generation. The packets in the captures are replayed based on time instead of the packet order. This means that the next packet from the capture is sent after a specified amount of time, ignoring whether or not the previous packet from the capture already arrived. The stateful mode can therefore be used to mimic stateful traffic without requiring the resources to track stateful connections on client and server side.

As we have previously described, we do expect to be able to measure all operations on the flow table with unidirectional traffic. Therefore, we still only generate traffic in one direction in the stateful mode. In contrast to the stateless mode, we generate each flow as a burst of packets that are sent quickly after each other. This probably matches the packet pattern of real connections better than the approach we chose with the stateless mode.

For our tests, we need to be able to configure the data rate r and the number of new flows f per second. The generated packets should all have the same size s. The number of packets to generate per flow, i.e., the packets in the PCAP file, can be calculated as $p_f = \frac{r}{sf}$. It can easily be seen that this number is not necessarily a integer value. We can not generate a fraction of a packet in the end of each flow. However, we only need to reach the number of packets per flow p_f on average over all flows. Therefore, we generate two PCAP files to replay with a length of $\lfloor p_f \rfloor$ and $\lceil p_f \rceil$ respectively. The two files are replayed a different number of times per second to achieve p_f packets per flow on average over the second. As the content of the replayed files is dependent on the configuration parameters of our test script, the files are generated in the setup phase of our script.

All our tests use UDP packets. To create different flows, TRex alters the source IP address and port of the packets read from the PCAP file. TRex replaces the original source with an IP address and port from a configurable pool. It with the lowest IP address and the lowest port and sequentially uses a new IP address until it reaches the limit of the IP address pool. Then the port number is increased by one and it starts again from the lowest IP address. By configuring pools of a sufficient size, we make sure that the flows will never be repeated.

In addition to the traffic from the PCAP files, TRex generates further packets to measure the latency. We configured these latency measurements to use ICMP packets and happen 1000 times per second, i.e., every 1 ms. The usage of ICMP ensures that the latency measurements will always reassemble a new flow, because the identifying five-tuple of a packet includes the protocol and is therefore different

for UDP and ICMP packets.

3.3.3 Improvements to TRex

TRex has some deficiencies that affect the analysis of the collected statistical data. First, all latency statistics are reported in a histogram with buckets arranged on a logarithmic scale. Therefore, the different buckets cover ranges of latency measurements of different size. More concretely, this means for example that in the range from 10 µs to 100 µs, each bucket covers 10 µs, while in the range from 100 µs to 1000 µs, each bucket covers 100 µs. This is a memory-efficient approach to analyze how many packets show a high latency. However, it is not feasible to collect an accurate latency distribution and determine the tail latency. Therefore, we apply patches from Massimo Girondi [20] that modify TRex to build latency histograms with a linear scale and equally sized buckets. In addition, we decrease the size of the buckets from 10 µs to 1 µs and set the lower limit of the histogram to 1 µs. The resulting histograms allow accurate analysis on the full range of the histogram.

Additionally, the main loop of the TRex server is modified. It is responsible for executing periodic tasks during the traffic generation, most notably the collection of statistics. The duration of the periods for the collection process increases noticeable over the intended 500 ms due to the increased histogram size which leads to increased execution times of the collection. The reason is that the main loop sleeps for a fixed amount of time after each execution of the loop. As the execution of the loop itself takes some time too, especially when collecting the big histograms, the periods increasingly differ from the intended periods. Therefore, we modify the main loop to sleep until a specific point in time using `clock_nanosleep` with `TIMER_ABSTIME`. With these modifications, TRex generates the statistics at an accurate period of 500 ms.

A similar modification is necessary in the TRex client. The statistical data are transferred to the client using a serialization in the JSON format. The serialized data needs to be deserialized at the client, which takes a long time due to the increased size of the histograms. Similar to the server, the client slept for a fixed amount of time after fetching one statistics update and its deserializing it. This leads to a few missed statistic entries over the course of a test run. The issue is resolved by modifying the client to sleep until a point of time after fetching the statistics like it is done at the server.

3.3.4 Integration in the testing application

Evaluation process requires the collection of different statistical data with various parameters (e.g, different data rates, different packet sizes, etc.). In order to automate the data collection, we developed a bash script to loop through the entire parameter space. For each parameter combination, we first start FastClick server, then initiate the traffic generation with TRex, and eventually store the statistical data into a separate file for later analysis.

3.4 Profiling with perf

Profiling entails observing the running software's called functions and the proportion of time they occupy the CPU. These data help us to analyze which operations might form a bottleneck on the performance of the load balancer. We use these results to tweak configurations for better performance, and to draw conclusions about the inherent weaknesses or advantages of certain flow classification methods compared to others.

In this report, we use perf for profiling the performance of LB. We limit its monitoring scope to the Click process and observe its specific performance. First, we use the TRex simulation to generate different types of data packets for testing, and we then run the load balancer. After TRex starts for five seconds, perf is triggered to record performance metrics for 30 seconds, which eventually generate an output file containing all measurement data. The 5-seconds delay ensures that FastClick is in the stable state of processing packets.

In order to understand FastClick performance from the perf output file, we use perf-class to classify each FastClick function in perf output into one function class that represents the corresponding load-balancing operation. For instance, the function `rte_hash_lookup` of the Cuckoo stateful load balancer is mapped to class `LoadBalancing-Lookup`, which represent the flow table lookup operation of the load balancer. The full list of function mapping is described in table Table 3.1. The column `perf-class regular expression` describes the list of regular expressions used in the perf-class mapping file to classify a function to a specific class [2].

After parsing the perf output in script format, perf-class produces the number of CPU cycles for each function class. However, due to the difference of number of packets each load balancer can process (e.g, stateless load balancer can process more packets than stateful load balancer), we can not directly use the CPU cycles to compare different load balancers. Instead, we divide the average CPU cycles per second of each function class by the average number of packets per second of

the corresponding load balancer and use this result to compare between the load balancers. For instance, assume that function class `LoadBalancing-Lookup` of the Cuckoo hash map load balancer spends 300 CPU cycles/packet, while the same function class of the stateless load balancer just occupies 150 cycles/packet, we can conclude that the lookup operation of the Cuckoo hash map load balancer is less effective than the stateless load balancer because it needs more CPU cycles to process one packet.

Function class	Description	perf-class regular expression
IO	Input and output operations	mlx5_rx_, mlx5_tx_, schedule, rxq_burtlrxq_cq, ToDPDKDevice, native_write, pmu_enable, RouterThread..Driver
NetworkProcessing	Functions related to networking process of the packet	Strip, Classifier, GetIPAddress, EtherEncap, BatchElement..push_batch, CheckIPHeader, ResetIPChecksum
LoadBalancing-Lookup	Lookup operation on the flow table	flowtable_lookup, rte_hash_lookup, rte_hash_bloom_lookup_data
LoadBalancing-Insert	Insert operation on the flow table	flowtable_insert, rte_hash_bloom_add_key_data, rte_hash_add_key
LoadBalancing-Process	Other minor tasks in FlowIPManager element	FlowIPManager
LoadBalancing-Logic	Main logic of the load balancer (i.e, perform modification of the packet to corresponding server)	FlowIPLoadBalancer
LoadBalancing-Concurrency	Lock and unlock operations in multi-core load balancers using mutex technique	pthread_rwlock_rdlock, pthread_rwlock_wrlock, pthread_rwlock_rdunlock, pthread_rwlock_unlock

Table 3.1: Classification of FastClick functions for perf-class

Chapter 4

Implementation

This chapter aims to provide an overview into the research conducted, in order to choose the additional flow classification method. It also details the implementations of the different flow tables using FastClick, for both single-core and multi-core configurations.

Firstly, Section 4.1.1 presents the different hash tables studied and a brief explanation on each of them. Secondly, Section 4.1.2 compares the previously stated methods using the specifications we have considered of importance in order to support our choice.

As previously mentioned in Section 2.1, there are two types of load balancers; stateless and stateful. We will implement one stateless load balancing method, described in Section 4.2.1, and multiple methods for stateful load balancers. Table 4.1 presents the different implementations for the stateful load balancer part of this project. For the case of a single-core configuration, we will implement the default flow table based on Cuckoo hash maps provided by DPDK, described in Section 4.2.2. We also implement flow tables based on C++'s unordered maps, in Section 4.2.3, and Hopscotch hash maps, in Section 4.2.4. Finally, we describe the implementation of Cuckoo++ in Section 4.2.5.

We will additionally scale these implementations to work on multiple cores using two different techniques. To begin, we leverage shared mutex locks to coordinate writing and reading by multiple threads to a single flow table, as described in Section 4.3.1. Then, Section 4.3.2 describes a different approach where all cores keep track of their own flow table as a way to scale operations.

	Single-core	Multi-core	
		Global Lock	*Per-core duplication*
Cuckoo	✓*		✓*
C++ unordered map	✓	✓	✓
Hopscotch	✓	✓	✓
Cuckoo++	✓		✓

* This was already implemented in FastClick

Table 4.1: Compared flow table implementations

4.1 Literature Review

4.1.1 Hash tables

As presented in Section 2.4.2.2, we will study different implementations of hash tables based on flow classification. Consider two prefetching situations. Firstly, in the pessimistic approach both buckets are prefetched to avoid late lookups. This is used by DPDK. Secondly, the optimistic situation, in which it is assumed that the data will be found directly in the primary bucket. This is used in MemC3 and CUCKOOSWITCH.

The first one implemented is Cuckoo from DPDK in Section 4.2.2, which uses five-tuple hashing to assign the element to two buckets, primary and secondary. This way entries only have to be looked up in these two buckets. On new insertions, previous elements may be relocated to their alternative option.

MemC3 [18] is an extension to Cuckoo, which stands for Memcached with Clock and Concurrent Cuckoo hashing. MemC3 uses the optimistic Cuckoo approach, where it is assumed that the key is found on the primary bucket, and so only that one is prefetched. Another improvement to Cuckoo is the reduction of overhead information by using timers, similar to the ones used for Cuckoo++, further explained in Section 4.2.5. These are used for late deletion, where entries are overwritten when expired timers are found in the bucket being accessed. Another similar extension of Cuckoo that uses the optimistic approach is CUCKOOSWITCH [52].

Cuckoo++ [42] is a hash table implementation that differs from Cuckoo by using a bloom filter that contains all keys that could not be added to the primary bucket, and so can be found in the secondary one. The main difference can be seen during lookup, where the key is searched within the filter before having to prefetch the secondary bucket. Thus reducing the amount of accesses to memory during this operation. Another difference is the incorporation of an expiration timer with

each entry, used for late deletion. Cuckoo++'s implementation is further described in Section 4.2.5.

Horton tables [8] are another extension of bucketized Cuckoo hash tables that distinguish between types A and B of buckets. While type A contains no extra information, buckets of type B include a remap entry that allows more items to be hashed with a single function. With this remap array all items that have overflown are kept track of. Maintaining a worse-case lookup of two buckets and reducing the majority of negative look-ups to one more bucket access. On the contrary, insertion is more complex specifically if the primary bucket is full.

Google's load balancer, Maglev [16], uses the connection tracking table to find a match, and if it exists then the connection is reused. While otherwise the new thread checks the consistent hashing module and selects a new server, adding the entry to the connection table for future packets with the same hashed five-tuple. When selecting a new server, five-tuple hashing is used and the new selection is added as a digest (rather than complete state information), which can later cause false hits.

Concury [44] is a very recent software load balancer that ensures no false hits and a low memory cost. By using a bloomier filter, it finds the specific destinations for the different incoming stateful packets and at the same time acts randomly for the stateless ones.

Hopscotch [31] is an algorithm that defines a neighborhood of size N and keeps the last location of the hashed key within its neighborhood. The location of that key can be moved inside that neighborhood to leave space for a more recent insertion by switching positions. Used in Section 4.2.4.

4.1.2 Comparison of reported performance numbers

The Cuckoo hash table can guarantee lookup in constant time but may require non-constant insertion time. Hence here, we list the previous different C++ implementations of hash maps, and summarize their performance based on existing benchmarks and evaluations [21, 42, 44]. Table 4.2 shows a comparison of the different implementations that we have considered for this literature review.

Cuckoo++ shows a better performance on the benchmarks than both optimistic, MemC3 and CUCKOOSWITCH, and pessimistic lookups for Cuckoo. It presents a 50% improvement in lookups when working on a single-core scenario and a 45% improvement when using 18 cores. To avoid most of the unnecessary accesses to the secondary bucket there is a less than 0.3% false positive rate with the use of the blooming filter. This rate indicates when the filter fails to correctly identify

whether the secondary bucket needs to be accessed, and so the secondary bucket is checked without the need to. It also requires less memory accesses.

Cuckoo++ needs additional metadata to store the bloom filter but has 50% less memory overhead due to the timers. Since Horton and Cuckoo++ have the same memory layout, Horton also reduces the memory overhead to 50%, but has a higher rate for false positives. Thus, Cuckoo++ performs better in these aspects. All the positive aspects of lookup that Cuckoo offers, worsen with insertion. The cost is higher, but is less important since lookups tend to take more execution time.

For insertion, Cuckoo++ has a small overhead due to the bloom filter and the timer, but can be neglected since it is very similar to DPDK's for large tables. When comparing Horton with Cuckoo++ [42], no distinction of types is used, and the tag is placed instead of the bloom filter. Horton has worse performance for all table sizes and worsens as the load factor increases, since the algorithm for insertion is more complex when the primary bucket is full. Horton tables also show worse lookup performance when working with higher load factors.

The multi-core evaluation shows similar results between Cuckoo++ and Horton for lookups, around 45% better than DPDK's Cuckoo.

Concury's memory cost is 20%-30% of Maglev's memory cost [44]. Compared to google:dense_hash_map, CUCKOOSWITCH presents worse performance when working with small tables [52], but overall better than dense_hash_maps for larger tables and always better than Google's sparse_hash_map.

The std::unordered_map, (using chaining method) works well in most cases [21], but is limited by the cache-unfriendliness of chaining. The tsl::hopscotch_map (using hopscotch hashing) and the tsl::robin_map (using linear robin hood probing) both are open addressing schemes like Cuckoo hash table, and have similar lookup speed for lower load factors. However, the former can provide a better compromise between speed and memory usage on higher load factors (>0.6). The tsl::robin_map has better insertion performance than Hopscotch. google::dense_hash_map (using quadratic probing) is not efficient in memory for higher load factors. Additionally, it performs poorly on lookup misses, just like the emilib::HashMap (using linear probing).

The tsl::sparse_map (using sparse quadratic probing) offers a good compromise between lookup time and memory usage, even when working with low load factors, although with slower speed on insertions. It is faster than both google::sparse_-hash_map and spp::sparse_hash_map. When using strings as keys, the tsl::array_-map (using array hash table) offers one of the best lookup speeds on large strings while having the lowest memory usage. However, its rehash process is slow and needs spare memory for copying maps. For larger objects, on insertion the values

may have to be moved around either because of the insertion process or because of rehashing. std::unordered_map and tsl::ordered_map (using linear robin hood probing with keys-values outside the bucket array) only need to move one element on deletion, and thus perform better.

In Table 4.2 we summarize the comparison found on different benchmarks and the papers of the implementations, with every algorithm in a separate row.

Later, Section 4.2.5 details which method we choose to implement and provides arguments to support this choice.

Name	Literature	Implementation	Insertion time	Lookup time	Memory efficiency	Multi-core
Cuckoo	yes [39]		=	=	=	=
MemC3	yes [18]			higher (than Cuckoo++)		
CUCKOOSWITCH	yes [52]			higher (than Cuckoo++)		
Cuckoo++	yes [42]	yes [43]	=	-50%	-50% (FPR <0.3%)	-45% (18cores)
Horton table	yes [8, 42]		+15%	+5%	-50% (FPR <5%)	-45% (18cores)
Concury	yes [44]	yes [47]		better	better	
Maglev	yes [16]			worse	worse	
CUCKOOSWITCH	yes [52]			better for larger tables	lower	
google::dense_hash_map		yes [27]		better for smaller tables	higher	
google::dense_hash_map		yes [27]	5th	5th	9th	
google::sparse_hash_map		yes [27]	9th	9th	1st	
tsl::sparse_map		yes [26]	7th	6th	3rd	
Hopscotch	yes [31]	yes [23]	3rd	3rd	6th	
tsl::robin_map		yes [25]	2nd	2nd	8th	
spp::sparse_hash_map		yes [41]	8th	7th	4th	
emilib::HashMap		yes [17]	4th	4th	7th	
tsl::ordered_map		yes [24]	1st	8th	5th	
tsl::array_map		yes [22]	6th	1st	2nd	

Table 4.2: Comparison of hash tables

4.2 Single-core load balancers

4.2.1 Round hashing (stateless)

The stateless load balancer is configured using the built-in FastClick `IPLoadBalancer` element [5]. The parameters of `IPLoadBalancer` include the virtual IP address of the load balancer and the list of IP addresses of the servers to handle the packet. For each incoming packet, `IPLoadBalancer` chooses one IP address from the list of servers and replaces the destination IP address field of the packet with this chosen IP address, thus the packet is forwarded to the corresponding server. All packets of the same flow will be forwarded to the same server. The FastClick processing flow of the round hashing is described earlier in Section 3.2.1.

4.2.2 Flow table using Cuckoo

The hash table load balancer is implemented using FastClick `FlowIPManager` and `FlowIPLoadBalancer` [3] elements as described in section Section 3.2.2

The `FlowIPManager` is the element in charge of the flow tables, where every incoming packet is evaluated and checked whether it is part of a new flow, or on the other hand, belongs to an existing one. If the new packet does not have the same key as one of the existing ones in the table, then a new entry is added to the table assigning a unique key to the new packet. This element uses the DPDK Cuckoo hash tables, where it maps each packet to a specific flow.

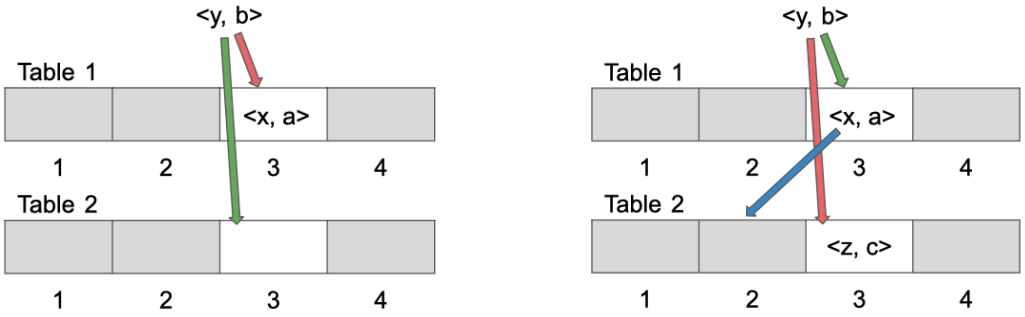
The `FlowIPManager` consists of two main functions which we will later modify to implement new elements using different hash tables. Those functions are *run_task* and *process*. The first one is in charge of freeing a certain position once a certain time has passed, and so there are no incoming packets from that specific flow for some time, leaving space for the assigned server to receive more packets from different flows. The *process* function receives a packet and checks whether that one already exists in the table, by using DPDK's *rte_hash_lookup*, the binary result refers to whether the flow-id of the packet exists on the table or not. If the packet belongs to a new flow then *rte_hash_add_key* adds the new flow-id to the table and checks if there has been an issue introducing the new entry on the table and resets the timers.

Cuckoo tables use flow classification to map each of the arriving packets to the flow it belongs to. This table again uses five-tuple hashing based on the fields of source and destination IP address, protocol, source, and destination port [15].

When given a flow table (array), a new hash is created with the same number of entries as the flow table. The hash key is set to the same number of bytes as the ones

on the flow key. This table is structured such that each entry is identified by a unique key, and so that all the keys have the same number of bytes as the time the hash is created. Also, the API contains a method that allows the `FlowIPManager` to work with the lookup entries in batches. This way, performance increases and so the function can work simultaneously with the next entries and with the current entries, reducing the amount of memory for accessing that is usually required.

The most useful part is that the Cuckoo hash resolves collisions. For any given key there are two buckets assigned (primary and secondary) to store that key in the hash table. Figure 4.1a shows an example of the case where one of the buckets is empty, and so the entry is added successfully (*green*). While in Figure 4.1b both buckets are full, and so one of the elements already in place has to be relocated to their other bucket option (*blue*), leaving space for the new entry (*green*). This way any entry only has to be looked up in two buckets for each lookup request. Thus, once the corresponding buckets are identified, each action to be made with that key (e.g., add, lookup) only has to be made within those buckets, reducing the lookup time.



(a) One of the buckets is empty

(b) The two buckets contain entries

Figure 4.1: Cuckoo hashing insertion example [28]

C++'s unordered maps (Section 4.2.3), Hopscotch (Section 4.2.4), and Cuckoo++ (Section 4.2.5) will be the three hash table algorithms implemented based on `FlowIPManager`, first for the given case of single-core and later on for multi-core configurations.

4.2.3 Flow table using C++ unordered maps

C++ unordered maps is a container that has key-value pairs with unique keys for each of the values and is included in the C++ standard library. This container allows to search, insert, remove and interact with many more functions with elements

from the tables [49]. As the name suggests, there is no particular ordering of these elements, they are placed depending on the hash of its key. The way of accessing the elements is then faster because once the hash has been computed it is very easy to refer to where the element is. The tests in this report use the implementation from *libstdc++* as bundled with *GCC 10.1.0*.

By replacing all the references to DPDK's hash table on the previous `FlowIP-Manager` element by functions of C++'s unordered maps we create a new element using a different table. For the new C++'s unordered maps element, we used *emplace* in order to insert a new element in the unordered_map and using the returning two values, the first one being an iterator pointing to the element inserted and the second one allowing us to see if it is a new flow or not. Then instead of the ones used by DPDK's Cuckoo hash tables, we use a faster access table.

To ensure that the size of the flow table does not grow, we reserve a table size and set a high maximum load factor, which for our current measurements should be unobtainable.

4.2.4 Flow table using Hopscotch maps

Hopscotch maps is named as such because of the hops that define the table's insertion algorithm. It uses a single n array of buckets, where each bucket has a neighborhood (collection of buckets at H positions) [31] [29]. The neighborhood size H should always be enough to hold all the items $log(n)$, but if the neighborhood is filled and there is no option to add the new entry, then the table is resized. Any given item will always be added or found in the neighbor of its hashed bucket. When a new element is added, it swaps some of the existing elements in order to maintain the distance H from the hashed bucket.

For a faster lookup, each bucket includes a bitmap with the information on which of the next $H-1$ entries contain an element which was hashed to the actual bucket. So that when an item is looked up, by checking that information, the algorithm can directly scan those referenced positions to find where the desired element is, instead of having to look at all $H-1$ buckets.

For example, in Figure 4.2 the size of the neighbourhood H is 3. The item 'c' has been hashed with value 4, but this position is occupied and the next free spot is in 8. Because position 8 is further than 3 positions from 4, the algorithm will swap one of the existing ones to another position. In our case, we see that the element 'b' (found at $H-1$) can be moved to the free position at 8 (since it is inside its neighbor). The new element is then inserted at the previous location of 'b'.

The new FlowIPManagerHopScotch files implement the Hopscotch maps so that

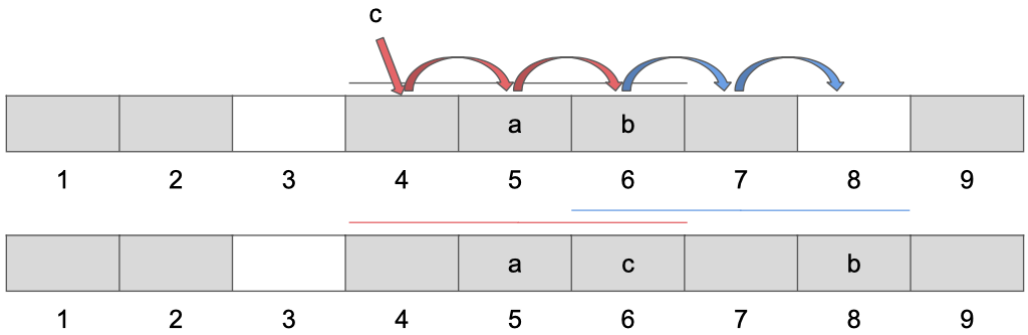


Figure 4.2: Hopscotch hashing example [31]

they can be used within our LoadBalancer. Adding the *include* directory to our path from [23], we can already use this Hopscotch library which replicates the behaviours used for C++'s unordered maps implementations with a few modifications.

The main difference with the previous implementation that we had to address are some limitations when using iterators. These were solved by, among others, calling the *value* function to access the second element on the pair when calling the function *emplace*. To prevent the flow table from growing, we use a growth policy with a high maximum load factor. We replace the existing *will_neighborhood_change_on_rehash* function so that it returns a static value. So when called from rehash to check if it would be of any benefit to rehash, it returns this value instead of failing.

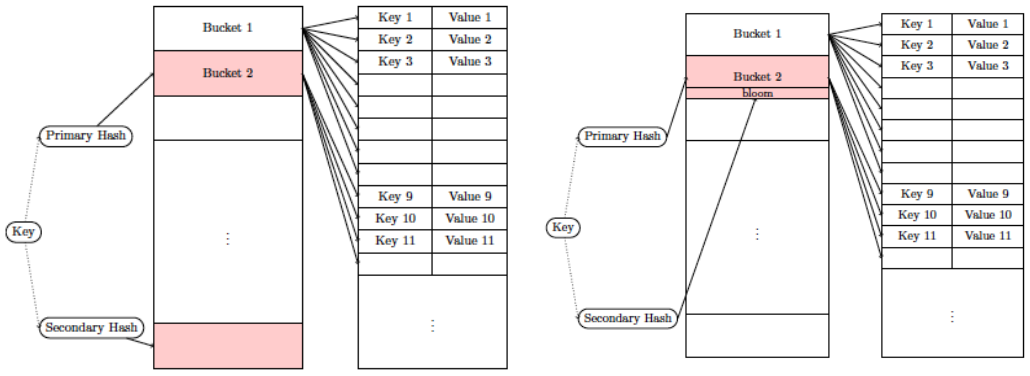
Similarly to Section 4.2.3, we set a fixed table size and add a maximum load factor of 32. We use 32 so that the maximum number of entries does not overflow when computing $table\_size * max\_load\_factor$, while still being enough to work with our measurements.

4.2.5 Flow table using Cuckoo++

Cuckoo++, as previously introduced, is similar to DPDK's default Cuckoo [15], previously described in Section 4.2.2, but offers better performance. Given the two previously introduced prefetching approaches, Cuckoo++ should give better results than both, especially when the entry is not found in the primary bucket. This is achieved by reducing the number of memory accesses to the secondary bucket when a lookup is done and by deleting elements only when an expired entry is accessed, instead of having to keep timeouts for deletion.

While Cuckoo by default accesses both buckets, this implementation uses a bloom filter that contains all the keys that could not be added to the main bucket, and so

had to be moved to the secondary one. Thus on lookup, the given key is searched in the bloom filter before accessing the secondary bucket, as can be seen in Figure 4.3 where the shaded slots indicate the accessed memory.



(a) Cuckoo

(b) Cuckoo++

Figure 4.3: Memory access during lookup on hash table [42]

A counter is also stored within this filter, which is incremented with each insertion to the secondary bucket and lowered with deletions for each of the elements stored in secondary buckets. So when the bloom filter counter reaches zero it is reset. This can give a few false positives since the key will still be in the filter while the element may not be there. The first advantage being that the amount of lookups in memory is reduced.

The second advantage is an attached expiration timer with each entry. Therefore, instead of deleting an element as soon as it expires, an element is only deleted when a new entry is inserted into its same slot and its timer has expired.

Cuckoo++ is the implementation that has been chosen after out literature review in Section 4.1. As just stated, the two main advantages identified that Cuckoo++ offers are reducing the amount of lookups in memory and an expiration timer associated with each entry. Though the second advantage has not been implemented since we decided not to work with timers and so to be able to give a fair comparison later with the other hashing algorithms. In Section 4.1.2 the reasons for this choice and the other hash tables that were considered are further detailed.

Cuckoo++ can be used in applications similarly as DPDK's Cuckoo, with some modifications [42]. The Cuckoo++ library was integrated into the FastClick build as part of the application itself, instead of a library that is linked later during the build. All references to previous DPDK's Cuckoo were replaced by changing some functions and removing unnecessary parts.

The functions names were changed by adding bloom to every previous rte_hash function, as defined on the GitHub repository [43]. Other functions used in other implementations such as *rte_hash_count* had to be changed by *rte_hash_bloom_size* in order to have the same result as the previous ones. Some functions return different outputs and expect different inputs than they did with DPDK's Cuckoo.

For lookup and insertion, the five-tuple hashing was stored in a key of type *hash_key_t* that was passed as a parameter. An increasing flow counter was also created to pass as an offset for insertions, making sure that it would never go beyond the table dimensions.

4.3 Multi-core load balancers

4.3.1 Shared mutex lock

Global locked flow table using C++'s shared locks uses a lock around the flow table. This protects the shared data from being accessed simultaneously by multiple threads. Multiple processing cores should not access the flow table at the same time in order to maintain integrity guarantees for the data. With this approach, if one thread has requested access to the flow table, then the other threads are blocked from accessing it until the first one has finished its operations.

The implementation utilizes C++ shared_mutex for concurrency control [48]. The shared mutex supports two types of locks: exclusive locks and shared locks. While only one thread can acquire the exclusive lock at a time, multiple different threads can acquire the shared lock simultaneously. Therefore, the implementation adopts shared locks for read access and exclusive locks for write access to the flow table.

Figure 4.5a describes the processing flow of this implementation. After receiving a packet, a thread will acquire the shared mutex lock to search the flow table to check if the packet belong to any existing flows. The shared lock is released as soon as the lookup operation is finished. In the following step, if there is no existing flow for the packet, the thread will acquire the exclusive lock to insert the new flow into the flow table. Eventually, the exclusive lock is released and the processing of the packet continues. This design allows multiple threads to perform lookup operations simultaneously as long as there is no thread writing to the flow table.

4.3.2 Duplication of the flow table across cores

Coordinating access to a shared flow table between threads provides significant overhead. Instead, the duplication approach creates a flow table for every thread,

which will be exclusively used by that respective thread. For example, when operating FastClick on four cores, there will also be four flow tables where a core stores all the flows it sees (Figure 4.5a).

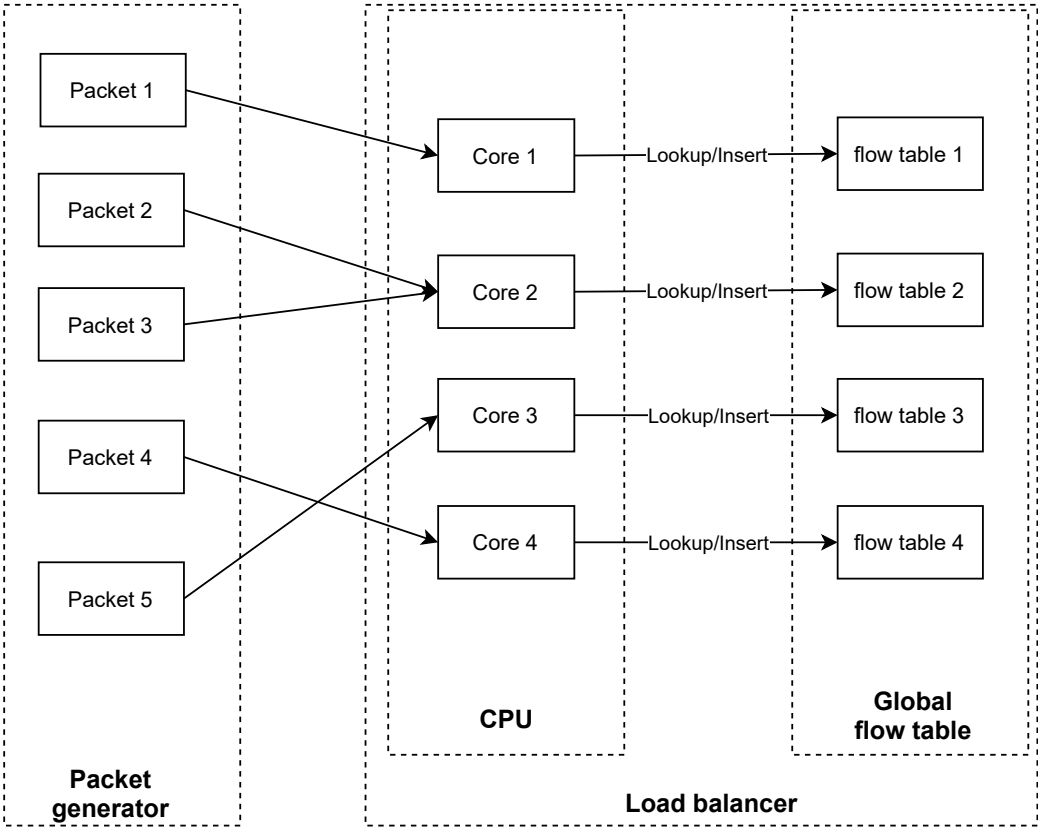


Figure 4.4: Per-core duplication technique in multi-core load balancer

Modern network cards use Receive Side Scaling (RSS) to distribute the incoming packets to different processor cores. It works similarly to a stateless load balancer in that it computes a hash for each packet over selected header fields, such as IP addresses or port numbers. Thereby, it ensures that packets, which belong to the same flow, are always directed to the same core. This means that a flow will only be recorded in one of the flow tables, which makes sure that there is only one selected backend server for each flow. Therefore, all the packets of a flow will still be routed to the same server.

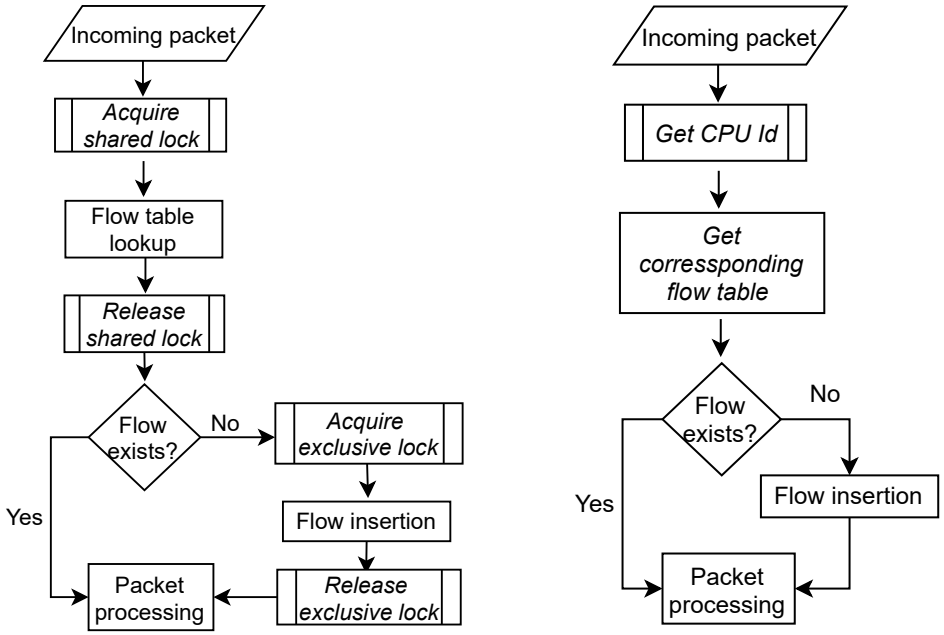
The implementation uses a pointer variable (global table) to store the list of flow table objects. Upon initialization, FastClick allocates the size of the global table based on the number of running threads which is retrieved from FastClick's `get_passing_threads` function. The total flow table capacity set in the

load balancer configuration will be equally distributed over all the cores' flow tables.

Figure 4.5b describes the processing flow of the per-core duplication technique. This implementation is similar to the single-core load balancer, with the exception that each thread reads and writes flow data into their corresponding flow table instead of shared table. In order to determine which flow table to use, each thread uses FastClick's `click_current_cpu_id` function to get their CPU id and access the global flow table list at the corresponding position.

To make full use of the independent flow tables per-core, it is necessary to ensure that no code of any FastClick element affected other cores. In general, the elements have no state that would require synchronization across cores. However, the `CheckIPHeader` element has a counter that is implemented as a `atomic_uint64_t`. This counter is incremented for every processed packet. As the increment operation is atomic, only one core can perform it at a given time while other cores, that try to increment the counter, are blocked. While these blocking periods are very short, they still significantly impact the performance.

Therefore, this counter is removed from the `CheckIPHeader` element. This is possible, because the counter is not needed outside the element in our configuration and it is only used for statistical reasons. Another approach would be to duplicate such counters per-core as well, but this is beyond the scope of this project.



(a) Mutex lock technique (b) Per-core duplication technique

Figure 4.5: Processing flows of multi-core load balancers

Chapter 5

Evaluation

This chapter presents two main evaluations of the load balancers in this project. Firstly, we evaluate and compare the two scaling methods — mutex-based scaling and per-core duplication — based on throughput results. Secondly, we evaluate and compare the five different hash table implementations based on throughput and latency results. The chapter begins by establishing the evaluation process.

5.1 Evaluation process

In order to evaluate our load balancer implementations, we measure their performance while sending synthetic network traffic using TRex. The TRex traffic generation parameter we vary is the data rate of outbound traffic. The values of this parameter varies depending on the test scenario. Our measurement script initiates the TRex network traffic with different values from our parameter space. Our parameter space consists of data rates starting from 10 Gbps up until 100 Gbps, with steps of 10 Gbps. Additionally, we vary the number of cores with which FastClick is run between 1, 2, 4, 6, and 8. The number of new flows generated every second is set to 10,000 and the packet size is set to 1000 bytes. The CPU frequency of the load balancer is configured to a fixed value of 1300 MHz.

5.2 Scaling methods

The first and most important comparison in this project is between different methods of scaling flow table operations across multiple cores. Figure 5.1 shows the achieved throughput of all scaling methods applied to all hash table implementations for different numbers of assigned CPU cores.

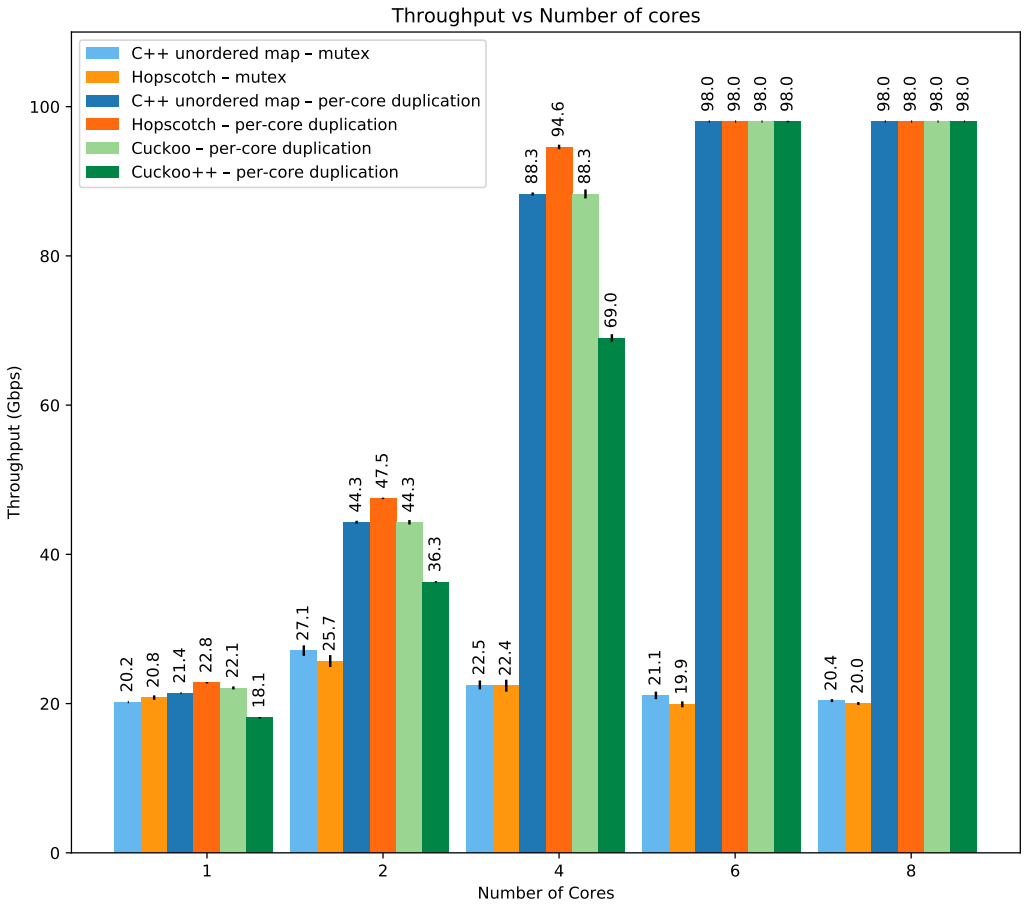


Figure 5.1: Throughput over different number of cores with error bars for multi-core load balancers after 5 samples, Data rate=100 Gbps

The main observation from Figure 5.1 is that mutex-based scaling performs worst. Doubling the number of cores from one to two achieves an improvement in throughput between 23% and 35%, but increasing the number of cores further leads to a decrease in throughput again. This performance plateau is explained by the overhead from operations related to concurrency. The mutex-based method exclusively locks the flow table when one core is performing insert operations on it, such that none of the other cores can access it at that time. This means that during such times, in multi-core situations, most of the cores (e.g, $n-1$ cores if FastClick runs with n cores) are waiting for their turn to perform any operations on the flow table, including lookups. This is the factor that slows down the mutex-based implementations, which Figure 5.2 clearly shows with the increase in computation cycles spent in concurrency related operations as the number of cores goes from two, to four, to eight.

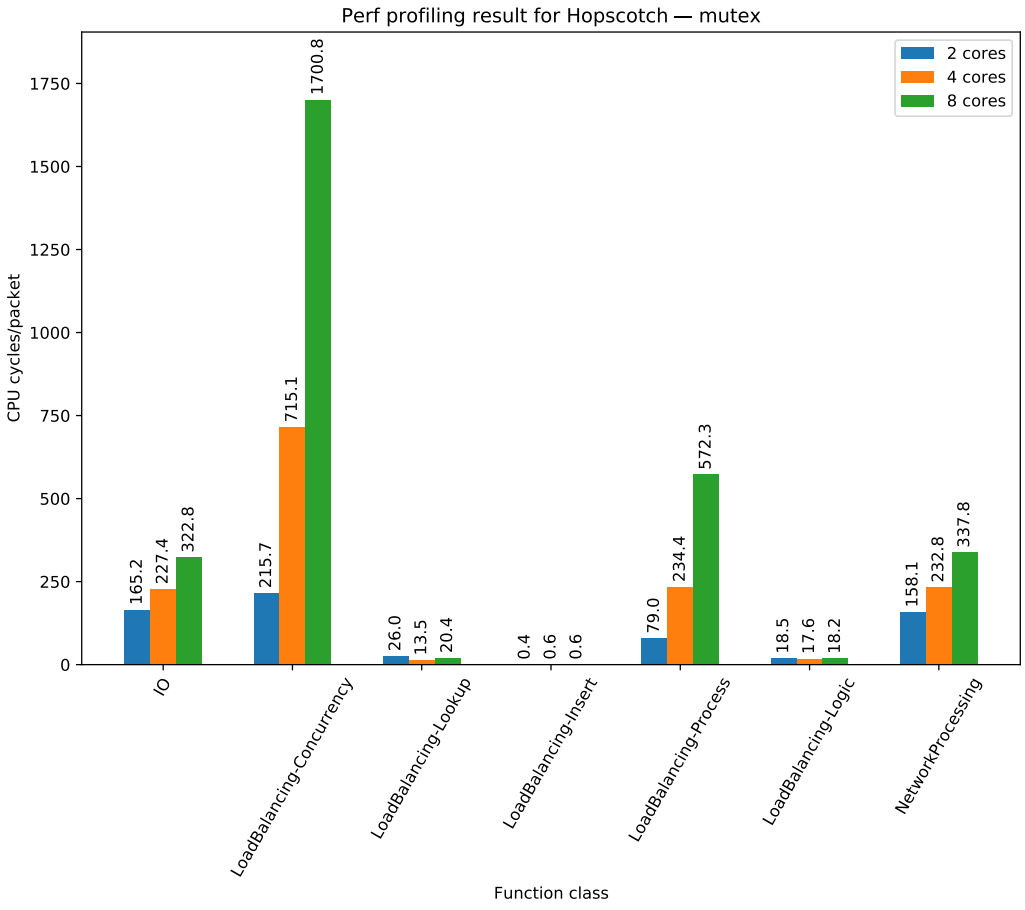


Figure 5.2: Perf profiling result of multi-core Hopscotch using mutex lock

Figure 5.3 also shows that mutex-based scaling leads to a high latency. While minor differences based on the hash table implementation can be observed, the latency appears to be dominated by the mutex-based scaling. This can as well be explained with the long periods where the cores are blocked and can not process further packets.

In contrast to the mutex-based method of scaling, the per-core duplication approach shows a linear relation between throughput performance and number of cores for all hash table implementations. With the results for *Hopscotch – per-core duplication* as an example, we observe performance improvements of 108% and 99% for doubling of the number of cores, from one to two, and from two to four, respectively. After going beyond four cores, the load balancers saturate the full link capacity, as can be seen for six and eight cores, where all hash tables using the per-core duplication method consistently achieve 98 Gbps of throughput.

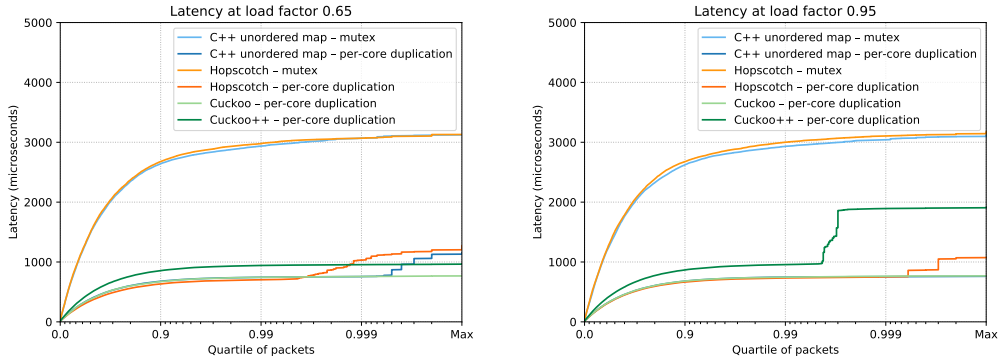


Figure 5.3: Latency comparison of different multi-core implementations at flow table load factors of 65% and 95%; based on 10,000 latency measurements; number of cores=4

We additionally expected the relative performance between C++ unordered map, Hopscotch, Cuckoo, and Cuckoo++ to remain stable. This hypothesis is confirmed by the data shown in Figure 5.1. Hopscotch consistently performs best in terms of throughput, followed by Cuckoo, C++ unordered map, and Cuckoo++. This order also holds for the latency of the majority of packets as can be seen in Figure 5.3. The significant performance difference between Cuckoo++ and the other hash tables is unexpected, and this will be further explored in Section 5.3.

In conclusion, the per-core duplication approach to scaling hash tables across multiple cores significantly outperforms the mutex-based approach. Using per-core duplication, the throughput performance improves linearly with the increase in number of cores. Mutex-based scaling reaches a performance plateau after which additional cores only cause decreased throughput.

5.3 Hash table performance

The second comparison is between the specific hash tables used to implement the flow tables. This section presents our test results and compares all hash tables and scaling methods based on the observed performance.

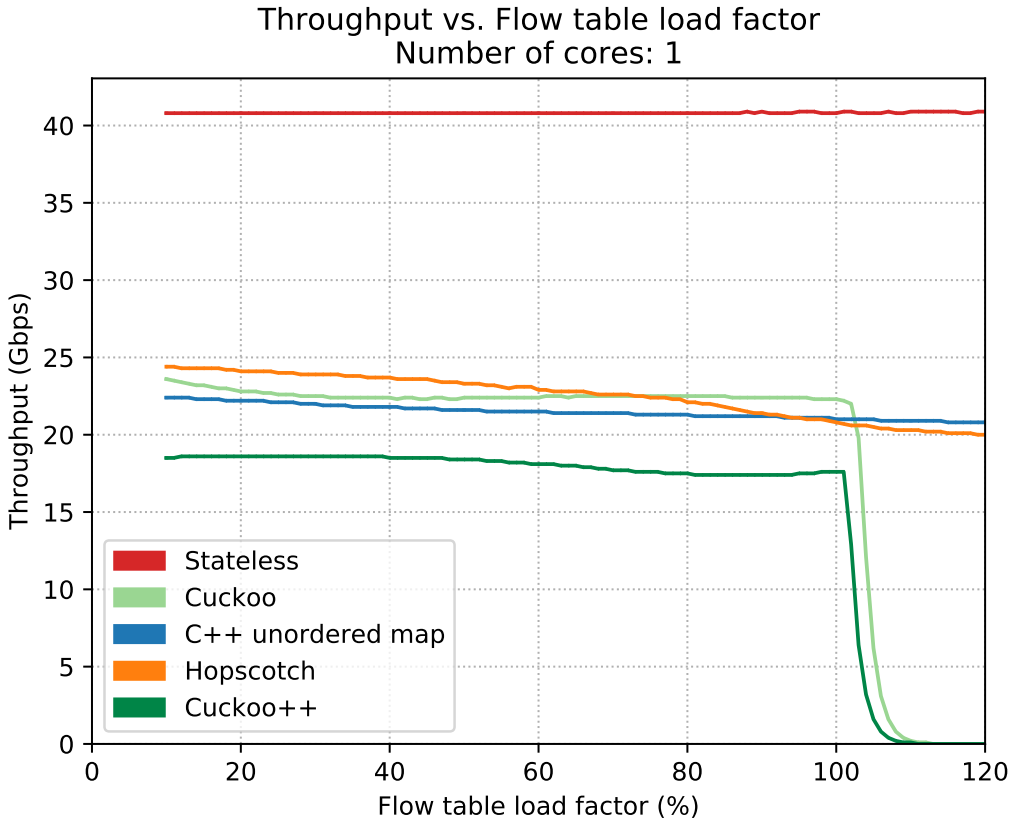


Figure 5.4: Throughput over increasing flow table load factor for single-core load balancer implementations after 5 samples, data rate=100 Gbps

Load balancers running on a single CPU core are the baseline of this comparison. Figure 5.4 shows the throughput for every hash table implementation over different flow table load factors. The stateless load balancer achieves a throughput rate more than 60% higher than the stateful implementations. This stems from the fundamental difference between these two load balancing approaches, namely that the stateful load balancers have the additional overhead of maintaining flow tables. Profiling the execution of all hash tables confirms this. Figure 5.5 clearly shows that the stateless load balancer spends zero CPU cycles on lookups, insertions, and processing, precisely because it does not maintain a flow table.

In Figure 5.4, it becomes clear that the Hopscotch hash table achieves the highest throughput up until a load factor of 70%, after which it decreases past the more stable Cuckoo hash table. The behavior of Cuckoo-based hash tables past a load factor of 100% is further explained in the following evaluation for the multi-core performance of all hash tables. The decreased performance of Hopscotch can

also be observed from the latency statistics. While the curve in the first graph in Figure 5.3 is below the other methods, it is shifted upwards to the same level as the other methods at a high load factor in the second graph.

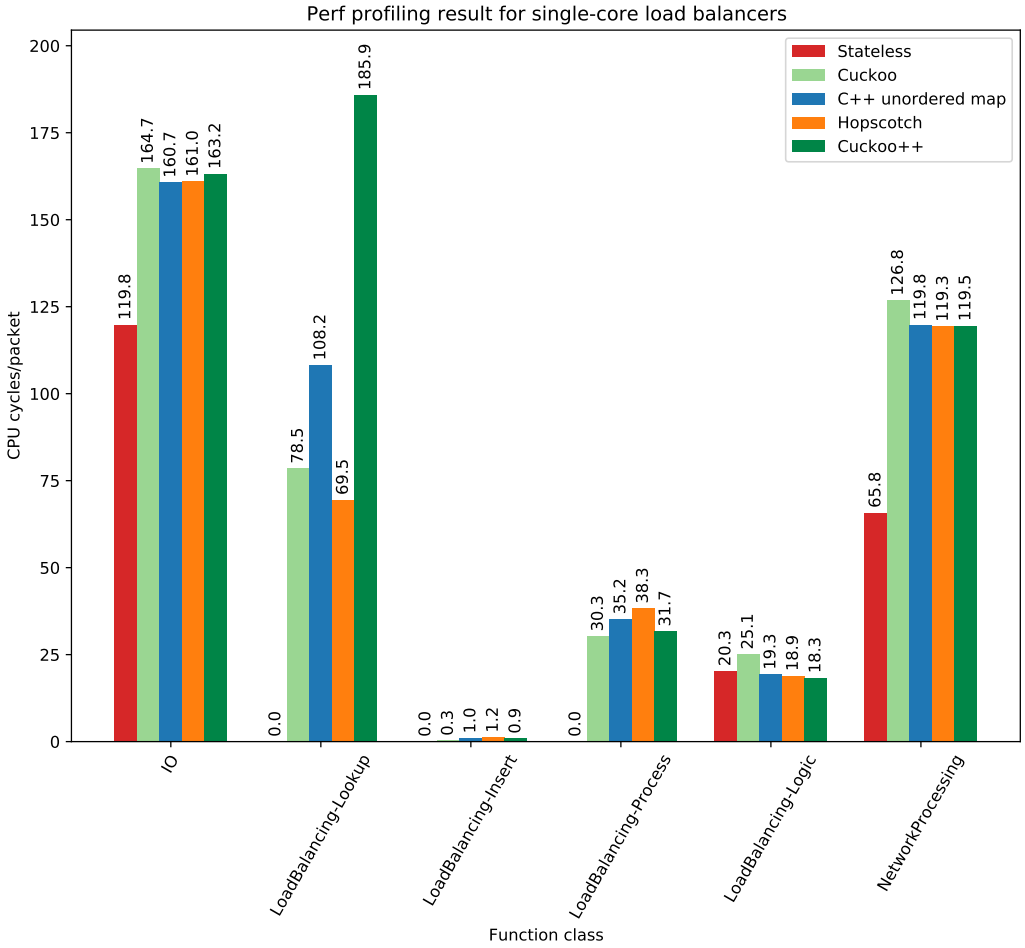


Figure 5.5: Perf profiling result for single-core load balancers, Data rate = 100 Gbps, each flow consists of 1250 packets of which only the first one triggers an insert operation

In Figure 5.6, the four graphs describe the achieved throughput for an increasing flow table load factor for different CPU core configurations. The horizontal axes continue until a load factor of 120%, because the tests were set up such as to observe how each hash table handles an overflow of the configured capacity and how it impacts performance. Figure 5.7 shows the achieved throughput against the data rate at which traffic was generated and sent to the load balancers. The following paragraphs provide details on several interesting conclusions that can be

drawn from these graphs.

Firstly, these graphs again confirm that the mutex-based approach to scaling across multiple cores impacts the throughput performance negatively, compared to the per-core duplication method. Irrespective of the flow table load factor, C++ unordered map and Hopscotch with mutex-based scaling consistently achieve throughput rates not higher than 27 Gbps.

In general, throughput performance of all implementations is stable throughout most flow table load factors. However, performance starts to change near a 100% load factor and beyond. Cuckoo-based hash tables show decreasing throughput after reaching a load factor of 100%. This is because these hash tables do not offer any overflow mechanism. Instead, they fail insertion and the load balancer then decides to drop the packet. In contrast, C++ unordered map and Hopscotch do provide a mechanism to store additional entries, albeit with worse performance. As the number of cores increases, the load factor at which Cuckoo-based hash tables start dropping in performance becomes lower, and with four and eight cores they even achieve some throughput at a load factor of 110%. Considering that there is no overflow mechanism, the latter observation is unexpected. However, we hypothesize that this behavior is caused by the manner in which the network card forwards network packets to the assigned cores. The network card distributes packets across cores by using a hashing algorithm. Depending on the hash functions that are used, this means each flow table might no longer produce a uniform distribution when applying its hash function on the entries to insert. In turn, this causes some cores and thus their flow tables to fill up quicker than others. When a flow table on one core is filled up, the Cuckoo-based methods will start refusing new insertions, causing the earlier drop in performance before the 100% load factor over all cores is reached. Similarly, it then also follows that other cores have space left for insertions, which causes the small amount of throughput in the tail of the graph.

The per-core duplicated Hopscotch implementation shows stable performance in all cases, but exhibits a slight decrease in throughput as the flow table load factor increases. The reason for this has to do with how Hopscotch works. As described in Section 4.2.4, Hopscotch performs swapping operations to maintain its neighborhoods. For higher load factors, Hopscotch will have to perform more swapping operations. These additional operations cause a drop in insertion speed, and thus a drop in achieved throughput. In general however, Hopscotch performs the best out of all the tested hash tables. A distinguishing feature of Hopscotch hash tables is their cache locality. This causes Hopscotch to require significantly less CPU cycles for hash table lookups, which Figure 5.5 confirms.

Cuckoo++ performs worst in our experiment, which is in contrast with the result

from research of the author, where it is more efficient than the default Cuckoo implementation in DPDK [42]. In order to investigate the cause of the performance difference, we investigated the profiling results of Cuckoo++. However, these did not reveal any obvious performance issues at the code level. The possible explanation for this includes: (1) the original research was conducted on DPDK 17.05, while our testbed is using DPDK 20.08, which may include additional optimizations for the default Cuckoo hash table that Cuckoo++ has not supported; (2) our use case is different from the use case of the original paper: while the original paper compares the two algorithms with data being stored on the table, our use case stores data outside of the table in a separate memory area; (3) Cuckoo++ is designed for very large hash tables that exceed cache sizes by trying to avoid memory accesses that would trigger cache misses. However, our hash tables does not reach this threshold and therefore there are not many cache misses that Cuckoo++ could avoid. We confirmed this hypothesis by using perf to record and compare the number of cache misses between Cuckoo and Cuckoo++ when they process data at 100 Gbps over a period of thirty seconds with the CPU frequency of 1300 Mhz. The perf output showed that Cuckoo++ has 65,000 cache misses, which is higher than Cuckoo at 60,000 events. As Cuckoo++ can not avoid cache misses in our test, the additional computation needed for the avoidance does only slow down the hash table lookups. Cuckoo is faster in our setting because it performs less operations for each lookup. While this applies to our test setting, the performance could be different if the flow table exceeds cache size according to the original research of Cuckoo++, where it benefits from the cache miss avoidance in comparison to Cuckoo.

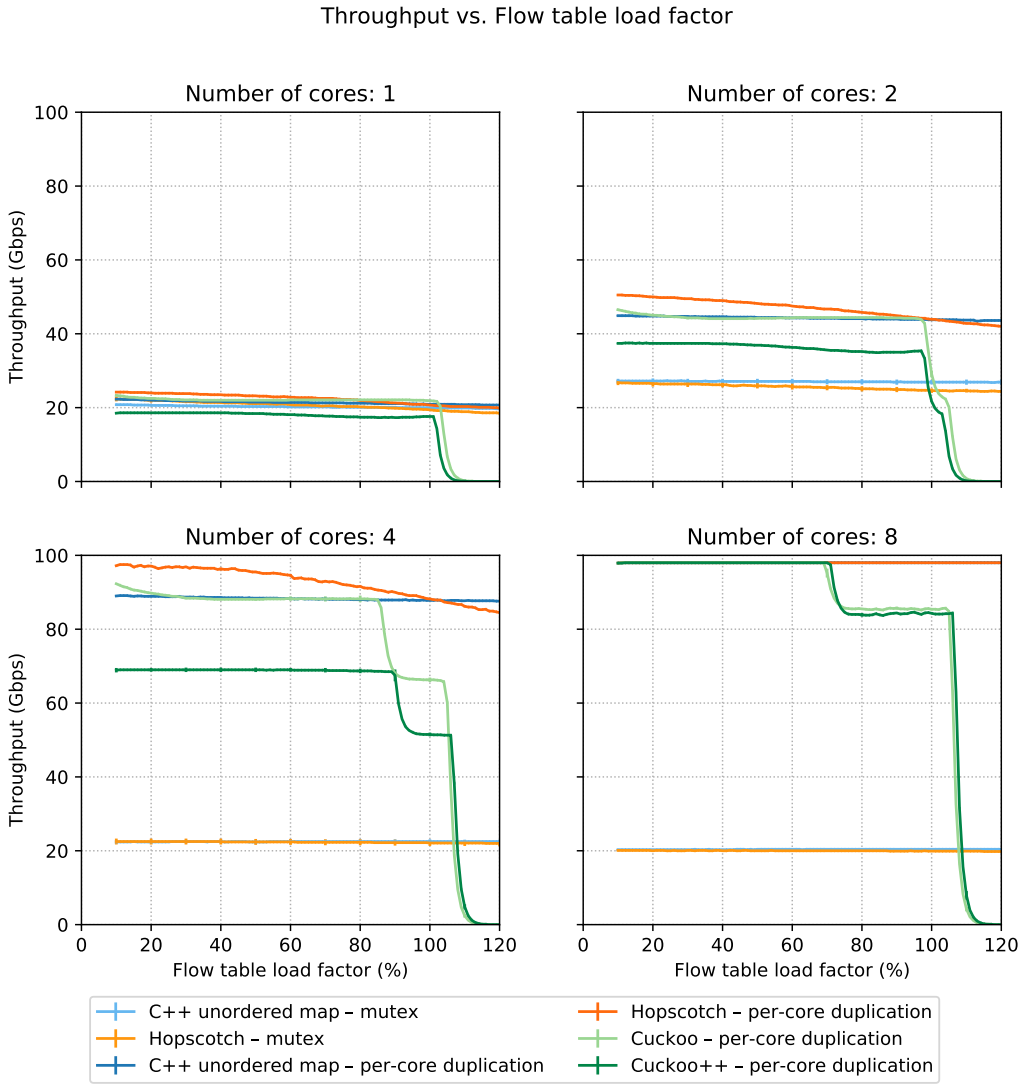


Figure 5.6: Throughput over increasing percentage of flow table usage for multi-core implementations after 5 samples, Data rate=100 Gbps

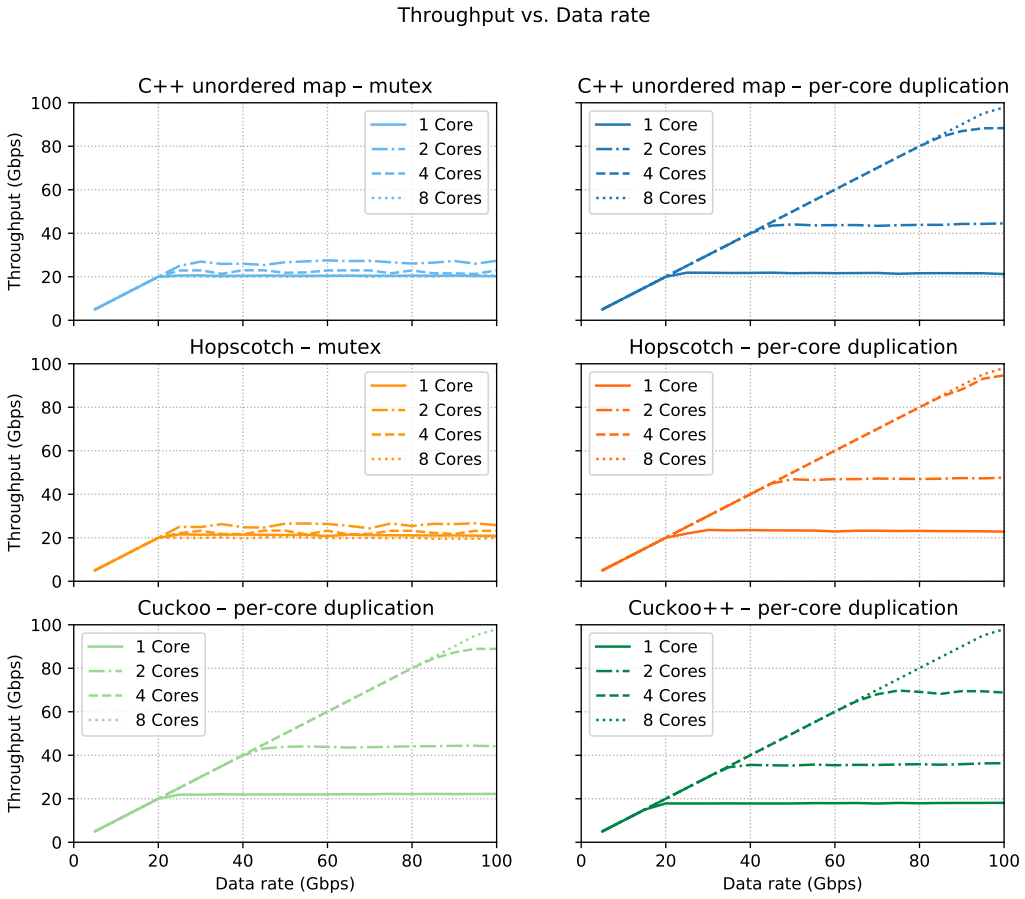


Figure 5.7: Throughput over different data rates and different number of cores for multi-core load balancer implementations

Chapter 6

Conclusion

The most important result from this project is the clear performance difference between the flow table scaling methods. The measurements and evaluation showed that the per-core duplication method outperforms the mutex-based method, irrespective of the specific hash table that is being used. Load balancers using the per-core duplication method of scaling, were observed to scale linearly in throughput performance as the number of cores increased. This contrasts with mutex-based scaling, where load balancers reached a performance plateau and subsequently were observed to decrease in throughput performance with the addition of CPU cores. In any case where the operation of flow tables is required to scale to multiple cores, we thus recommend to use the per-core duplication method.

This project additionally compared several hash table implementations, including Cuckoo, C++ unordered map, Hopscotch, and Cuckoo++. Our results and evaluation showed that the differences between hash tables are less clear than the observed distinction between scaling methods.

In general, Hopscotch was shown to perform best across all our test cases. At higher load factors, Hopscotch's performance decreases slightly, but not as drastically as Cuckoo-based hash tables.

Unexpectedly, Cuckoo++ did not perform better than Cuckoo during our measurements. Its achieved throughput was mostly lower than the throughput achieved by Cuckoo. Our profiling revealed that Cuckoo++ spends significantly more CPU cycles on lookup operations. This performance difference is partially explained by the relatively small, configured capacity of the hash tables we used. Cuckoo++'s authors note that the method becomes more efficient for much larger tables. Our project insufficiently tested this. However, we recommend that enterprises with relatively small tables, serving a few hundred thousand clients, do not

use Cuckoo++ with their load balancers, based on the results we observed in our measurements.

Because the performance differences between hash tables were observed to be marginal, potential future users should also take other features into consideration. An advantage of the Cuckoo++ implementation are its built-in timers and expiration mechanisms. DPDK's implementation of Cuckoo includes lock-free operations, that can be used to maintain a shared state for other purposes across cores.

Depending on specific usage scenarios of a load balancer, there are more factors to include in a comparison such as the one conducted in this project. For example, with specific numbers for the amount of flows per second to support, the measurement setup as described in this report can be used to accurately test performance of all hash tables in the respective scenario. This project thus contributes by not only providing a general comparison of hash tables and scaling methods, but also by enabling future comparisons to take place for more specific usage scenarios.

6.1 Future work

There are several direct ways to take the work from this project further. Firstly, DPDK 18.11 introduced extendable buckets, which provides an overflow mechanism for Cuckoo. Performing all the tests again with an updated DPDK installation, is thus expected to produce better results for the default Cuckoo hash table. The release notes mention a 100% insertion success [14].

Secondly, it became apparent that we have not tested Cuckoo++ with hash table capacities at which it was designed to perform better. In future work, including those cases in the measurements might result in Cuckoo++ proving itself more effective.

Lastly, we have not considered memory efficiency of any of these hash tables during the project. Future work can include this metric in the comparison, which might shift the final relative results.

References

[1] A10 Networks, Inc. *Application Delivery and Load Balancing: Thunder ADC | A10 Networks*. URL: https://www.a10networks.com/products/thunder-adc/ (visited on Jan. 5, 2021).

[2] Tom Barbette. *{tbarbette/Perf-Class}: Python Parser for Linux Perf Profiler to Group Symbols into Classes*. URL: https://github.com/tbarbette/perf-class (visited on Jan. 2, 2021).

[3] Tom Barbette. *FlowIPLoadBalancer Element Documentation*. May 7, 2020. URL: https://github.com/tbarbette/fastclick/wiki/FlowIPLoadBalancer (visited on Oct. 5, 2020).

[4] Tom Barbette. *FlowIPLoadBalancerReverse Element Documentation*. May 7, 2020. URL: https://github.com/tbarbette/fastclick/wiki/FlowIPLoadBalancerReverse (visited on Oct. 5, 2020).

[5] Tom Barbette. *IPLoadBalancer Element Documentation*. May 7, 2020. URL: https://github.com/tbarbette/fastclick/wiki/IPLoadBalancer (visited on Oct. 5, 2020).

[6] 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 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS). Oakland, CA, USA: IEEE, May 2015, pp. 5–16. ISBN: 978-1-4673-6633-5. DOI: 10.1109/ANCS.2015.7110116.

[7] Tom Barbette et al. "A Low-Level Dive into Building a High-Speed NFV Dataplane for Service Chaining". In: (Apr. 24, 2018), p. 2.

[8] Alex D. Breslow et al. "Horton Tables: Fast Hash Tables for in-Memory Data-Intensive Computing". In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, June 2016, pp. 281–294. ISBN: 978-1-931971-30-0. URL: https://www.usenix.org/

conference/atc16/technical-sessions/presentation/breslow.

[9] Cisco. *Cisco Annual Internet Report (2018–2023)*. 2020. URL: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.pdf.

[10] Cisco. *TRex - Realistic Traffic Generator*. URL: https://trex-tgn.cisco.com/ (visited on Oct. 3, 2020).

[11] Citrix. *What Is a Load Balancer?* Citrix.com. URL: https://www.citrix.com/glossary/load-balancing.html (visited on Oct. 6, 2020).

[12] Miyuru Dayarathna, Yonggang Wen, and Rui Fan. "Data Center Energy Consumption Modeling: A Survey". In: *IEEE Communications Surveys & Tutorials* 18.1 (Jan. 27, 2016), pp. 732–794. ISSN: 1553-877X. DOI: 10.1109/COMST.2015.2481183.

[13] Constantinos Dovrolis, Brad Thayer, and Parameswaran Ramanathan. "HIP: Hybrid Interrupt-Polling for the Network Interface". In: *ACM SIGOPS Operating Systems Review* 35.4 (Oct. 2001), pp. 50–60. ISSN: 0163-5980. DOI: 10.1145/506084.506089.

[14] DPDK Developers. *DPDK Release 18.11 - Release Notes*. URL: https://doc.dpdk.org/guides-18.11/rel_notes/release_18_11.html (visited on Jan. 6, 2021).

[15] *DPDK: Hash Library*. URL: https://doc.dpdk.org/guides/prog_guide/hash_lib.html (visited on Oct. 14, 2020).

[16] Daniel E. Eisenbud et al. "Maglev: A Fast and Reliable Software Network Load Balancer". In: *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Mar. 2016, pp. 523–535. ISBN: 978-1-931971-29-4. URL: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eisenbud.

[17] Emil Ernerfeldt. *Emilk/Emilib: Loose Collection of Misc C++ Libs*. URL: https://github.com/emilk/emilib (visited on Nov. 23, 2020).

[18] Bin Fan, David G. Andersen, and Michael Kaminsky. "MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, Apr. 2013, pp. 371–384. ISBN: 978-1-931971-00-3. URL: https://www.usenix.org/

conference/nsdi13/technical-sessions/presentation/fan.

[19] Rohan Gandhi et al. “Duet: Cloud Scale Load Balancing with Hardware and Software”. In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM ’14. Association for Computing Machinery, 2014, pp. 27–38. ISBN: 978-1-4503-2836-4. DOI: 10.1145/2619239.2626317.

[20] Massimo Girondi. *TRex Patches for Linear Latency Histograms*. Nov. 2019. URL: https://github.com/MassimoGirondi/trex-core/compare/cf462ff77defbab902eed80f30050214b398dc5f...9801e5b6850dd5692eaf1e79d26b36c0c6068ba6.patch (visited on Jan. 5, 2021).

[21] Thibaut Goetghebuer. *Benchmark of Several Hash Maps*. URL: https://tessil.github.io/other/hash_table_benchmark.html (visited on Nov. 19, 2020).

[22] Thibaut Goetghebuer. *Tessil/Array-Hash: C++ Implementation of a Fast and Memory Efficient Hash Map and Hash Set Specialized for Strings*. URL: https://github.com/Tessil/array-hash (visited on Nov. 18, 2020).

[23] Thibaut Goetghebuer. *Tessil/Hopscotch-Map: C++ Implementation of a Fast Hash Map and Hash Set Using Hopscotch Hashing*. Oct. 26, 2020. URL: https://github.com/Tessil/hopscotch-map (visited on Oct. 23, 2020).

[24] Thibaut Goetghebuer. *Tessil/Ordered-Map: C++ Hash Map and Hash Set Which Preserve the Order of Insertion*. URL: https://github.com/Tessil/ordered-map (visited on Nov. 18, 2020).

[25] Thibaut Goetghebuer. *Tessil/Robin-Map: C++ Implementation of a Fast Hash Map and Hash Set Using Robin Hood Hashing*. URL: https://github.com/Tessil/robin-map (visited on Nov. 18, 2020).

[26] Thibaut Goetghebuer. *Tessil/Sparse-Map: C++ Implementation of a Memory Efficient Hash Map and Hash Set*. URL: https://github.com/Tessil/sparse-map (visited on Nov. 18, 2020).

[27] Google. *Sparsehash/Sparsehash: C++ Associative Containers*. URL: https://github.com/sparsehash/sparsehash (visited on Nov. 23, 2020).

[28] Emmanuel Goossaert. *Cuckoo Hashing*. July 20, 2013. URL: http://codecapsule.com/2013/07/20/cuckoo-hashing/ (visited on Nov. 29, 2020).

[29] Emmanuel Goossaert. *Hopscotch Hashing*. Code Capsule. Aug. 11, 2013. URL: http://codecapsule.com/2013/08/11/hopscotch-hashing/ (visited on Nov. 2, 2020).

[30] Pankaj Gupta and Nick McKeown. "Packet Classification Using Hierarchical Intelligent Cuttings". In: *Proc. Hot Interconnects VII* (Aug. 1999).

[31] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. "Hopscotch Hashing". In: *Distributed Computing*. Ed. by Gadi Taubenfeld. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 350–364. ISBN: 978-3-540-87779-0. DOI: 10.1007/978-3-540-87779-0_24.

[32] Eddie Kohler et al. "The Click Modular Router". In: *ACM Transactions on Computer Systems* 18.3 (Aug. 2000), pp. 263–297. ISSN: 0734-2071, 1557-7333. DOI: 10.1145/354871.354874.

[33] P Ramakanth Kumar and N Deepamala. "Design for Implementing NetFlow Using Existing Session Tables in Devices like Stateful Inspection Firewalls and Load Balancers". In: *Trendz in Information Sciences & Computing(TISC2010)*. Computing (TISC). Chennai, India: IEEE, Dec. 2010, pp. 210–213. ISBN: 978-1-4244-9007-3. DOI: 10.1109/TISC.2010.5714641.

[34] Xiaozhou Li et al. "Algorithmic Improvements for Fast Concurrent Cuckoo Hashing". In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. Amsterdam, The Netherlands: Association for Computing Machinery, 2014. ISBN: 978-1-4503-2704-6. DOI: 10.1145/2592798.2592820.

[35] H. Lim et al. "A Quad-Trie Conditionally Merged with a Decision Tree for Packet Classification". In: *IEEE Communications Letters* 18.4 (Apr. 2014), pp. 676–679. ISSN: 1558-2558. DOI: 10.1109/LCOMM.2014.013114.132384.

[36] Hyesook Lim, Min Young Kang, and Changhoon Yim. "Two-Dimensional Packet Classification Algorithm Using a Quad-Tree". In: *Computer Communications* 30.6 (2007), pp. 1396–1405. ISSN: 0140-3664. DOI: 10.1016/j.comcom.2007.01.004.

[37] Zhi Liu et al. "BitCuts: A Fast Packet Classification Algorithm Using Bit-Level Cutting". In: *Computer Communications* 109 (Sept. 2017), pp. 38–52. ISSN: 01403664. DOI: 10.1016/j.comcom.2017.05.001.

[38] Loadbalancer.org. *Hardware Load Balancer*. URL: https://www.loadbalancer.org/products/hardware/ (visited on Jan. 5, 2021).

[39] Rasmus Pagh and Flemming Friche Rodler. “Cuckoo Hashing”. In: *Journal of Algorithms* 51.2 (May 2004), pp. 122–144. ISSN: 01966774. DOI: 10.1016/j.jalgor.2003.12.002.

[40] *Perf: Linux Profiling with Performance Counters*. URL: https://perf.wiki.kernel.org/index.php/Main_Page (visited on Oct. 5, 2020).

[41] Gregory Popovitch. *Greg7mdp/Sparsepp: A Fast, Memory Efficient Hash Map for C++*. URL: https://github.com/greg7mdp/sparsepp (visited on Nov. 23, 2020).

[42] Nicolas Le Scouarnec. “Cuckoo++ Hash Tables: High-Performance Hash Tables for Networking Applications”. In: *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*. ANCS ’18. Association for Computing Machinery, 2018, pp. 41–54. ISBN: 978-1-4503-5902-3. DOI: 10.1145/3230718.3232629.

[43] Nicolas Le Scouarnec. *Technicolor-Research/Cuckoopp: A High-Performance Hash-Table for Network Packet-Processing Applications (Compatible with DPDK)*. URL: https://github.com/technicolor-research/cuckoopp (visited on Nov. 17, 2020).

[44] Shouqian Shi et al. “Concury: A Fast and Light-Weight Software Cloud Load Balancer”. In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. SoCC ’20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 179–192. ISBN: 978-1-4503-8137-6. DOI: 10.1145/3419111.3421279.

[45] Oleksii Shyshkov. *Load Balancing*. July 20, 2018. URL: https://oshyshkov.com/2018/07/20/load-balancing/ (visited on Oct. 2, 2020).

[46] Sumeet Singh et al. “Packet Classification Using Multidimensional Cutting”. In: *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM ’03. Association for Computing Machinery, 2003, pp. 213–224. ISBN: 1-58113-735-4. DOI: 10.1145/863955.863980.

[47] *Source Code of the Concury Prototype*. 2019. URL: https://www.dropbox.com/s/ruou2l340uu1f4u/concury%20code.zip (visited on Nov. 16, 2020).

[48] *Std::Shared_mutex - Cppreference.Com*. URL: https://en.cppreference.com/w/cpp/thread/shared_mutex (visited on Nov. 16, 2020).

[49] *Std::Unordered_map - Cppreference.Com*. URL: https://en.cppreference.com/w/cpp/container/unordered_map (visited on Oct. 13, 2020).

[50] Liu Tianhua et al. "The Design and Implementation of Zero-Copy for Linux". In: *2008 Eighth International Conference on Intelligent Systems Design and Applications*. 2008 Eighth International Conference on Intelligent Systems Design and Applications (ISDA). Kaohsuing, Taiwan: IEEE, Nov. 2008, pp. 121–126. ISBN: 978-0-7695-3382-7. DOI: 10.1109/ISDA.2008.102.

[51] TRex team. *TRex Stateless Support*. URL: https://trex-tgn.cisco.com/trex/doc/trex_stateless.html (visited on Oct. 11, 2020).

[52] Dong Zhou et al. "Scalable, High Performance Ethernet Forwarding with CuckooSwitch". In: *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*. CoNEXT '13. Association for Computing Machinery, 2013, pp. 97–108. ISBN: 978-1-4503-2101-3. DOI: 10.1145/2535372.2535379.