Hardware Offloading for Connection Tracking

Gruppen

Amanda Nee, Andreas Kinnunen, Emilia Blidborg
Johan Edman, Raman Salih, Vignesh Purushotham Srinivas

IK2200

Communication Systems
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden

10th January 2022

Examiner: Dejan Manojlo Kostic

Abstract

Load balancing is an important part of the Internet infrastructure's scalability. This is of even higher importance now as the performance increase in computer chips is declining at the same time as the traffic amount on the Internet is increasing and is expected to continue to do so. In combination with the move to Network Function Virtualization, to get flexibility, cheaper and more available commodity hardware, ways to increase performance are needed. The first step was to optimize software, but to make the Central Processing Unit utilization even more efficient Hardware Offloading is now investigated. This paper investigates the benefits of using Hardware Offloading for the Load Balancer's connection tracking. Throughput and latency are compared with and without Hardware Offloading, to illustrate how the two differ. The results show that offloading connection tracking to hardware could be beneficial. While there is an initial impact on performance when inserting a large number of flows, there is much performance to be gained. Additionally, if flows are selectively offloaded the initial impact can be mitigated to some extent, and the benefits increased. More investigation into the threshold for the selective offloading is required, with the potential for additional performance improvements.

Keywords

Load Balancing, Hardware Offloading, Smart NIC

Acknowledgements

The authors of this report would like to thank Dejan Kostic and Massimo Girondi for their support and guidance during this project.

Contents

List of Figures

List of Tables

List of Acronyms and Abbreviations

ASIC Application Specific Integrated Circuits

BF Basic Forwarder

CPU Central Processing Unit

DIPs Direct IPs

DPDK Data Plane Development Kit

DUT Device Under Testing

HO Hardware Offloading

IoT Internet of Things

LB Load Balancer

LPM Longest Prefix Matching

NF Network Function

NFV Network Function Virtualization

NIC Network Interface Controller

PMD Poll Mode Driver

RSS Receive Side Scaling

SDG Sustainable Development Goals

SDN Software Defined Networking

TG Traffic Generator

VIP Virtual IP

Chapter 1

Introduction

With the world becoming more connected every day, an ever-increasing amount of devices are connecting to the Internet and generating vast amounts of traffic. Traffic that often traverses the globe and must be directed to its destination over the various interconnected routers and switches that make up the Internet, while also being able to keep guarantees for latency and minimal loss of packets. Having an effective and adaptable network infrastructure that is capable of handling the traffic without introducing large delays or losses is therefore increasingly important. A statement that can be put into perspective by the predictions made by Cisco [1], a prediction that the number of Internet users will rise to 5.3 billion (66 percent of the global population) by 2023 - an increase of 1.4 billion from 2018. Beyond that, it is forecasted that an additional 14.7 million connections will arise from machine-to-machine communication alone due to the increase of connected Internet of Things (IoT) devices.

This can be expected by the fact that almost $20 billion is estimated to be spent by companies during 2021 on the IoT sector alone [2]. Thus, the need for making the underlying network infrastructure sustainable and effective is one of the biggest challenges being faced going forward. One way to deal with a large number of requests is through using Load Balancers (LBs), where the strain that would be put on a single server is spread out over many serving the same content. However, as traffic increases in volume and speed, having an effective mechanism for balancing the traffic becomes paramount as to not have the LB itself become a bottleneck and adversely affect its clients.

1.1 Problem Statement

Network Function Virtualization (NFV) and Software Defined Networking (SDN) are two ways to reduce cost and allow faster service deployment by decoupling network functions from expensive specialized hardware through virtualization. It allows the function to instead be run on more flexible, and cheaper, off-the-shelf commodity hardware. The specialized hardware has always had the benefit of having more performance than general-purpose hardware, but this has changed. Up until recently, computer chips have been increasing in performance at an exponential rate while the price has not. This, however, ended around 2005 Fig. 1.1 when the clock frequency of computer chips stagnated, forcing chip manufacturers to find other ways to improve performance. For example, by the use of accelerators or innovative chip designs with more cores that have turned out to be more expensive [3]. But also through software techniques that have seen limited scalability and the need for specific software design.

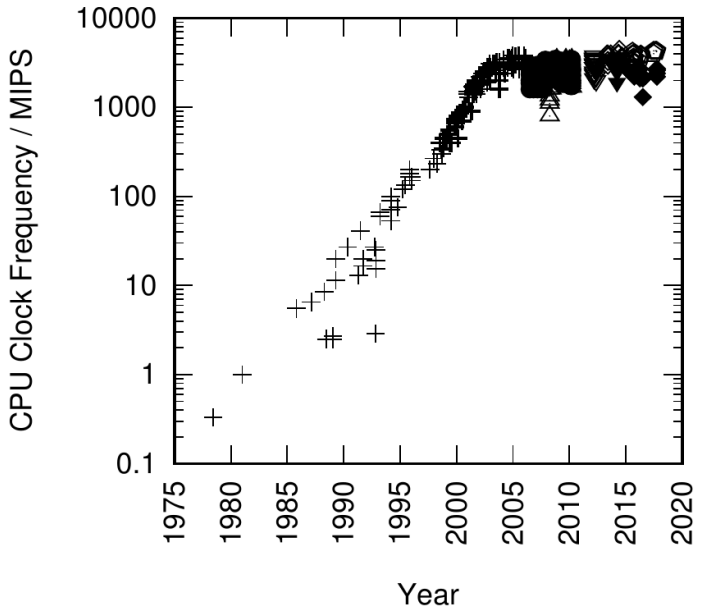


Figure 1.1: Clock-frequency/MIPS of Intel Central Processing Unit over time shown in log scale. Showing the stagnation of frequency increase around the year 2005.

source: Is Parallel Programming Hard, And, If So, What Can You Do About It? [4].

This creates a problem for the use of commodity hardware as buying new and faster hardware is becoming a less viable solution to the problem of meeting increasing performance demands. A way to mitigate the problem, to a certain degree, has been to attempt to make the software more efficient. One such solution is to bypass the kernel in the system and allow the code to execute solely in userspace. Therefore, execution does not have to wait for the relatively slow and expensive kernel calls and scheduler, or its interrupt-based network code. But this is not enough and more efficiency is still sought to make use of existing hardware and the flexibility it provides.

Load balancing is a popular candidate for NFV, especially in SDN. It is also a computationally demanding function requiring lots of resources to perform well. Connection tracking is an essential functionality in LBs, which is used to differentiate flows. This is necessary to maintain coherent packet flows for the network services and applications that are being served. While the Central Processing Units (CPUs) installed on the LBs are flexible and can handle such tasks, the ever-increasing network speeds and volume of individual packets is increasing at a higher rate than the computational power in commodity hardware does. Thus commodity hardware will not be able to handle future network demands.

1.2 Goals

The goal of this project is to evaluate the effectiveness of hardware offloading for connection tracking in LBs. The project will focus on developing and testing different techniques to offloading connection tracking to hardware with the ambition of achieving better utilization and maximizing its effectiveness.

Additionally, a literature study will be performed looking into state-of-the-art

offloading techniques for network flows in LBs.

1.3 Research Questions

The research question is to determine if offloading the connection tracking to hardware is an effective mean to improve the performance in a load balancing scenario.

1.4 Research Methodology

By referencing the *Portal of Research Methods and Methodologies*, the project falls under the category of quantitative research [5]. The project will take an analytical approach using small data sets. Through statistical analysis of gathered data, it will be possible to support or reject the hypotheses and draw conclusions to answer the research question. The project will consist of developing two components in FastClick; a Traffic Generator (TG) and a LB. While the LB is the main interest, the TG serves the role of benchmarking the LB configurations. By varying the kind of traffic generated and amount, statistics on the LB's performance can be collected. The statistical analysis of the collected metrics can then determine how well the LB performs in its configuration. By automating the benchmarking it will be possible to test different parameters and determine what implications each parameter has. With the automation, it will also be possible to ensure that results are reproducible. The project itself will be divided into stages, each leading to a more advanced LB configuration.

Stages

1. Development of the basic components, to be able to measure baseline performance
   (a) The Traffic Generator - Used to benchmark the LB
   (b) The Load Balancer - The component to be benchmarked
   (c) The Basic Forwarder (BF) - A component for recording baseline performance
2. Implement offloading of connection tracking to hardware
3. Optimize the flow rules used in the connection tracking

1.5 Delimitations

This study is limited to a specific time scope, and the full feature set that the LB provides will not be considered. The study will be limited to using a round-robin LB for the execution.

Furthermore, the traffic generator will not generate traffic on its own, but replay traffic from trace files that have been obtained from Caida [6]. While these traces contain a multitude of traffic, the project will be limited to IP packets, and exclude other types of traffic. As most traffic in the real world consists of IP packets, and the traces used are captured on public routers, the results should still be generally applicable and reasonable as a realistic scenario.

1.6 Sustainability and Ethical Reflection

Due to the increased amount of connected devices and users on the Internet, along with the projected increases in traffic and power consumption [1], [7], the demands on the underlying network infrastructure will increase as well. It is clear that neither horizontal nor vertical scaling will be environmentally sustainable in the long end, especially if the Sustainable Development Goals (SDG) are to be considered [8], as power demands would continue to rise along with the added hardware resources and machines. One solution is to be able to more effectively use the hardware that is already in place.

This project aims to investigate if Hardware Offloading (HO) would increase the effectiveness of LBs. If so, it would allow more clients to be served from a single LB, thus reducing the need to further expand with additional hardware and defer future purchases of more powerful hardware to the future. As the HO is a feature that is built into the network card, it would not require any additional hardware and replacements of network cards could be made to gain the potential performance benefits. There could then, however, be a concern for increased amounts of E-waste [9], should this prove an advantageous feature. The hope, in the long run, would be that more effective hardware, with increased performance and capability, instead lowers the hardware requirements for a given performance goal. Ultimately then lowering the hardware demand, manufacturing requirements, operational power demands and subsequently lowering E-waste.

Thus, if the overall goal of increasing performance and by extension lowering power consumption, the demands on natural resources are lowered, and the waste material produced - which can be connected to several sustainability goals [8], [9].

The authors of the project do not feel that there any direct ethical consequences that could be seen arise as a consequence of any results found in this project. Mainly due to the fact that the project is technically oriented and not focused on social sciences.

1.7 Structure of The Report

First, a review of the background to the problem is presented in Chapter 2 with information regarding the relevant components, such as LB, software stack, and flow classification. Thereafter, details regarding the methodology, testbed and implementation are described in Chapter 3. The results are then presented in Chapter 4 along with an analysis in Chapter 5. Finally, conclusions regarding the results and reflections on the overall outcome of the project are detailed in Chapter 6, together with closing words on what future work could be performed in the area.

Chapter 2

Background

Network Function Virtualization (NFV) is the practice of decoupling the Network Functions (NFs) from the hardware and by doing so being able to use any general hardware configuration for a task, while the software can stay largely unchanged. This gives great benefits in flexibility but loses out on some of the specialized hardware's performance.

NFV is employed to allow for better flexibility and scaling in network infrastructure. Examples of commonly virtualized NFs are network security systems, such as firewalls and IDS, accelerators, and LBs. However, as network bandwidths continue to increase, the CPU that is generally doing most of the heavy lifting in packet processing, starts to become a bottleneck - and processing packets at line rate has become a difficult task in most network systems, leading to it becoming a highly researched area [10]–[12].

NFV has allowed the decoupling of NFs from proprietary hardware, enabling the use of more generally available, cheaper, commodity hardware. This results in better resource utilization, flexibility, and adaptability, but is not perfect in all its aspects. With the ever-increasing network speeds, performing the NF and handling packets at line-rate with only the raw processing power of the CPU has

stopped scaling with the end of Moore's Law and Dennard Scaling [3].

A common way to deal with this, and reduce the load on the CPU is to implement the functionality directly in custom silicon. Specific tasks such as compression, cryptography, and Longest Prefix Matching (LPM) can then be offloaded from the CPU and performed in specialized hardware. The main benefit of this is that the CPU does not have to be involved to such a large extent, if at all. In many cases, it often leads to computations being able to be performed at a higher rate as well.

One virtualized NF of particular interest in today's Internet is load balancing. A LB has the task of spreading incoming network packets to multiple destination servers so that they are all, to the best extent possible, evenly loaded. In stateful load balancing, the system must keep track of each connection established to assure that packets are forwarded correctly and that flows remain coherent. Load balancing and connection tracking have proved to be an expensive task [10], [13], [14] with the risk of being bound by the CPU's processing capacity at high network speeds. Network interfaces with the support to perform such features have recently become available, but it remains to see how effective they are at offloading the CPU [15]–[17].

2.1 Load Balancing

LBs are ubiquitous network components that are crucial for keeping today's Internet functioning. The content/application providers deploy the LBs capable of serving millions of connections per second to thousands of clients [18].

LBs provide translation from Virtual IP (VIP) to Direct IPs (DIPs). In data centre environments, services are implemented on multiple dedicated servers and assigned unique DIPs, but the service provider exposes only one or a few VIPs to

the outside world. Any client trying to access the service then addresses requests to the exposed VIP. A LB then would receive a request on VIP and route it to one of the DIPs. Thus, a LB processes every request intended for a particular service. This high traffic volume introduces heavy load on both the data and control plane of the LB [13]. As a result, the throughput of the LB directly influences the overall performance of the underlying service. A LB must therefore provide high performance and availability.

Additionally, a LBs main objective is to ensure uniform load distribution and connection consistency [18]. *Connection consistency* is defined as the ability to always forward the packets belonging to the same connection or session to the same end server. *Uniform Load Distribution* ensures that no single end server is overwhelmed with service requests. LBs can be classified into two main categories depending on how they distribute the traffic flows, namely stateless and stateful.

2.1.1 Stateless Load Balancers

Stateless LBs are often faster, less complicated, and have reduced computational cost compared to *stateful* LBs. The LB looks at the hash of the 5-tuple of the packets and uses that to determine which of the end servers to send the packet to, this means that the same client is always connected to the same end server. The limitation with this approach is by simply using a hash algorithm to compute which server to send the packets to there is no certainty that the distribution of the packets on the end servers will be executed uniformly since it does not consider the actual amount of load the end servers each are under since it treats every request equally. It also has the additional limitation of being difficult to scale up since the hashing algorithm will need to be changed to reflect the new LB setup.

2.1.2 Stateful Load Balancers

Stateful LBs, also called dynamic LBs works by checking if the packets are coming from a certain client and then routing them all to the same end server. By looking into the current workload of each end server the LB can pick different servers to distribute the load uniformly. Packets originating from new clients are assigned to servers while packets sent from clients that have previously sent a packet is routed to the same end server the first received packet was assigned, the LB will therefore remember the states of each client [19]. These ”session states” are saved in a database called the flow table. Therefore for every arriving packet, the stateful LB must perform a lookup on the flow table to check if the flow is new or previously seen and also check if an end server has been assigned to it, thus it is important to consider how these new connections are handled.

The limitation with this approach is that it is less efficient since more operations are being executed per packet, first, it is hashed just as in the stateless LBs then you need to check if it is present in the flow table, if not it needs to be inserted in it as well, and then finally route the packet to the end destination. These operations regarding the flow table lead to higher latency and lower throughput. Additionally the ”expired” flows needs to be removed from the flow table and the cost of this operation is an additional limiting factor on the performance [20].

2.1.3 Hardware-based Load Balancers

Hardware LBs are often implemented on custom hardware chips to accelerate processing. These chips are tailored to provide application-specific load balancing at a very high rate.

The primary limitation of the hardware LB is cost-efficiency during scaling. Typically, these LBs are “racked-and-sacked” within a data centre, hence, an

operator must procure more compatible hardware LBs to scale. Additionally, incorporating third-party hardware in data centres reduces custom feature roll-out and testing. With the rise of commodity servers and high-speed network cards, the lines between hardware load balancing and software load balancing are increasingly disappearing [10]. ty

2.1.4 Software-based Load Balancers

Software LBs are implemented using software running on commodity servers equipped with high-speed network cards. Software load balancing provides better scalability, higher availability and is more cost-efficient. Additionally, a virtualized LBs allow faster iterations to test and deploy new versions, which is prohibitively time-consuming, and not easily done without the assistance from the manufacturer on hardware-based LBs [10].

Software LB with their increase in freedom of what functions to implement. And how to implement these functions they suffer from the need to send the packet to be handled in software. Thus they first have to send the packet incoming on the hardware to a memory space readable from the general-purpose CPU, and then back to the hardware to send the packet on its way after processing. These extra steps add a time delay before the packet can be forwarded from the time it arrived at the hardware.

2.1.5 Hybrid Load Balancers

Due to the limitations of hardware LBs being costly to implement and software LBs often lacking the latency that the custom Application Specific Integrated Circuits (ASIC) chips provide [13], a mix of these two implementations is sometimes used. As the name implies hybrid LBs utilizes both methods together

at the same time. The hard part of utilizing this implementation is knowing what flows to offload to the hardware and what flows to keep on the software side. By offloading to the Network Interface Controller (NIC) several advantages are gained in different ways. For instance, by using the NICs on-board computational power, the load of the CPU is reduced, which leads to an increase in the capacity that the LB can handle. Additionally, one would generally achieve lower latency since the processing of the packets is done in fewer processor cycles [12]. Hybrid LBs therefore achieve a combination of the advantages of software (higher flexibility) and hardware (better performance).

2.2 Software Stack

This section focuses on presenting FastClick and Data Plane Development Kit (DPDK). Fastclick can be configured to enable fast I/O with DPDK as the underlying packet framework, to achieve high throughput and low latency.

2.2.1 Data Plane Development Kit

DPDK is a set of libraries and drivers designed to accelerate packet processing for high-throughput applications [21]. Created by Intel in 2010 and subsequently made available as open-source, it has been maintained under the Linux Foundation's involvement ever since. The main technique that enables the fast packet flow for DPDK is bypassing the kernels and its standard networking stack by using Poll Mode Drivers (PMDs). PMDs consist of APIs using userspace drivers to configure the RX and TX devices and their respective queues.

In traditional processing of data packets, the system is relying on interrupts to the CPU, i.e. the network card receives the data packet and informs the CPU, which causes an interrupt, the CPU then computes the data and delivers it to

the appropriate protocol stack. However, if the amount of data passing through the network card is large, this way of handling packets will constantly issue the CPU interrupts, which causes higher latency and requires the need for more CPU cores [22].

With the use of DPDK kernel bypass is achieved by creating a fast path from the NIC to the application within userspace, and thus completely bypassing the kernel as the name implies. With PMD running, the NIC does not cause interrupts to the CPU upon received packets, instead, PMD constantly polls the NIC for packets to be processed. In this way, the user plane applications, e.g. FastClick, can process the I/O almost instantly. This way of processing packets will consume more CPU time due to the need of having to wait for packets. However, for latency-sensitive applications, this approach is necessary. The alternative would be to have the system flooded with interrupts to deal with new packets directly, which effectively could leave the system unable to deal with any previous packets or other tasks.

2.2.2 FastClick

FastClick [23] is a high-speed userspace packet processing framework that evolved from Click Modular Router [24]. Click introduces a flexible and modular software architecture for building routers by using packet processing modules called elements, but evolved into a NFV development platform. Click abstracts the NFV development by allowing users to compose graphs of elements. Elements are units of simple networking functions. Packets are then routed from the source elements to destination elements of the graph while being modified by the intermediate elements.

When compared to Click, FastClick added support for DPDK and enhanced the Netmap Integration. FastClick achieves higher processing speed over Click by employing the following modifications:

- Full push paths
- I/O Batching for computation
- Multi-queue architecture to exploit Receive Side Scaling (RSS) and multi-core processing
- Zero packet copy by using pre-allocated packet pools

A simple forwarder implemented with FastClick on modern hardware can process packets at a rate of 80Gbps (with packet length of 1200 bytes) on a single core [25].

2.3 Flow Classification

RFC 3697 [26] defines a flow as a sequence of packets from a particular source to a particular unicast, anycast, or multicast destination. Essentially, a flow consists of all the packets belonging to a specific transport connection. Traditionally, flow classifiers relied on the 5-tuple (source IP address, destination IP address, source port, destination port, and transport protocol) to compute flow identifiers, however, flow identifiers can also be computed from any protocol stack using addresses and packet attributes [27].

Flow-aware stateful LBs classify packets into different flows based on the 5-tuple. Packets from the same flow are forwarded to the same back-end server. While hardware-assisted connection tracking for load balancing is likely to gain performance, not all flows have an equal impact on performance. As the

flow tables and rule capacity in the network card are limited [28], the design of the rules to select flows to offload must be chosen with care to maximize the performance gains - if any. There are numerous characteristics of flows that can be considered while developing flow selection techniques. The flow offloading selection algorithm should maximize the overall performance and prevent hardware flow tables from overflowing. It could, for example, be more beneficial to offload only “heavy hitters” or “long-lived” flows in comparison to “mice” and “short-lived” flows.

The following subsections briefly describe a selected few flow offloading techniques that have been implemented in the project.

2.3.1 Naïve Offloading

The ’naïve offloading’ principle involves offloading the first n flows that arrive at the LB. While this technique yields the least amount of overhead in terms of packet processing, it cannot distinguish any characteristics of the flows, and therefore, all flows are treated the same. However, this will effectively favour heavier flows, as they occur more commonly on the Internet traffic [29].

2.3.2 Heavy and Light Hitters

An elephant flow is a flow that is characterized by having traffic size and duration that is significantly higher than other flows. Most of the Internet’s traffic (around 80%) is carried by a small number of flows (elephants), while the remaining large amount of connections are very small in size or lifetime (mice) [29]. Thus, the impact of elephant flows on network performance is significant. Identifying and offloading elephant/mice flows that are overwhelmingly consuming network resources might allow for further performance improvements.

Accurate flow characterization is a complicated process, mostly due to scalability issues. Storing per-flow level state adds resource overhead in already constrained network devices [30]. This work employs a rather simpler threshold-based approximation method to classify flows. For each active flow, counters that track the number of packets, size, and duration are maintained. Once the counters exceed a certain threshold T, the flow is classified as a heavy hitter (elephant flow).

2.3.3 Fastest Flow First

While the above offloading strategy selects flows based on flow size, this technique focuses on packet rate. A flow with a higher packet rate contributes a larger fraction of packets that traverse the LB at any given time. Hence, offloading such flows can potentially lead to a greater reduction of processing time in the software. Therefore, every packet from an offloaded flow skips the hash computation and flow table lookup, which can be a taxing operation [12].

2.4 Related Work

Load balancing comes in many types and implementations, such as strictly software LBs, hardware-based LBs, and also hybrid LBs [12]. Traditionally, load balancing is implemented by having dedicated hardware. However, this implementation strategy does have some issues. For instance, dedicated hardware LBs are complex to deploy, costly, and difficult to scale up/down based on the demanded load in the network [31].

This is what Google is addressing with the state-of-the-art software LB Maglev [10]. It is a large distributed software system that can be deployed on commodity Linux servers. Unlike traditional hardware LBs, its capacity can be easily adjusted by adding or removing servers. To accommodate the high and ever-increasing traffic loads, Maglev is optimized for packet processing performance. It does this by utilizing consistent hashing to load balance the traffic across service instances to minimize the negative impact of unexpected faults, changes, and failures in the network.

Despite the lower cost and better scalability/flexibility of the strict software LBs compared to the traditional hardware LBs, they suffer from higher latency and lower throughput compared to the strictly hardware-implemented approaches [32]. This is why a hybrid implementation is sometimes preferred. The hard part of utilizing this implementation is knowing what flows to offload to the hardware and what flows to keep on the software side, which is what Raphael Durner and Wolfgang Kellerer aim to answer [33]. They concluded that deciding which packet flows to offload to the hardware using the machine learning algorithm J48 has the best properties of all the investigated machine learning algorithms. J48 is a Java implementation of the C4.5 algorithm presented by Quinlan, J Ross in the book *programs for machine learning* [34]. The J48 algorithm is based on a decision tree generated from the training data, which will be used for the classification of the different flows. It combines a high offloaded data rate with low table occupation and additionally has the lowest flow classification complexity. Other algorithms like RandomForest and NaiveBayes were also tested and showed great potential as well since they are only slightly worse and could also be used efficiently by the LB.

Chapter 3

Method and Implementation

This chapter aims to describe the method used during the project. First, the testing environment is presented, then the implementation of the applications are explained and lastly, the testing procedure is described including how the data collection is performed.

3.1 Testing Environment

The two servers used were running Ubuntu 18.04 with kernel version 5.4. The upstream version of FastClick (commit e2033ca) were used in combination with DPDK version 21.08. They are also both equipped with an Intel(R) Xeon(R) Gold 5217 CPU, coupled with 48 GiB of RAM each and together with an NVIDIA Mellanox ConnectX-5 100 Gbps Ethernet [35] NIC for each machine.

The project measured the performance on a physical (non-virtualized) testbed, running the tests on the same machines with identical settings. The testbed consists of two machines interconnected by a 100 Gbps fibre link. One machine, the TG, generated and received the traffic, simulating clients from the Internet and the receiving load-balanced servers. The other machine, the Device Under Testing

(DUT), were deployed as a layer 3 LB to test the various LB configurations, or as a BF for measuring baseline performance. Since the network link is full 100 Gbps duplex, i.e. supports bidirectional traffic at speeds up to 100 Gbps, the same link was used for both sending the data being generated from the TG as well as receiving it back from the DUT without affecting the generality of the results. All parts are implemented in FastClick which were configured using DPDK, polling, and flow API. The setup is visualized in Fig. 3.1 and 3.2.

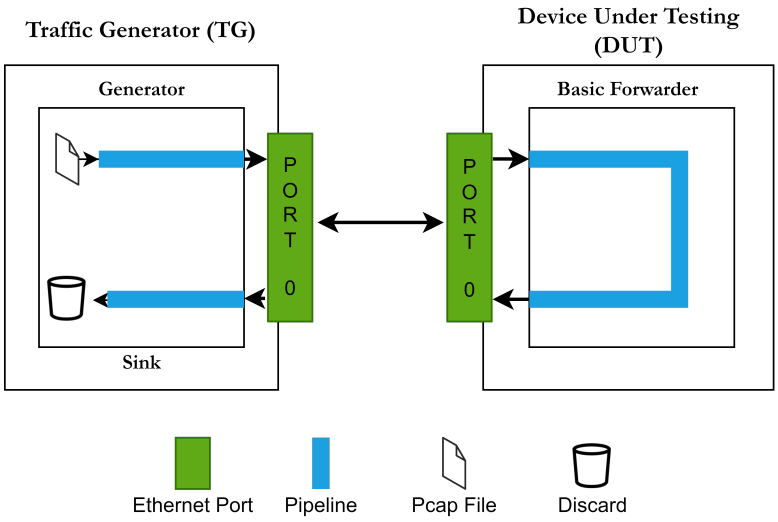


Figure 3.1: Hardware topology Basic Forwarder

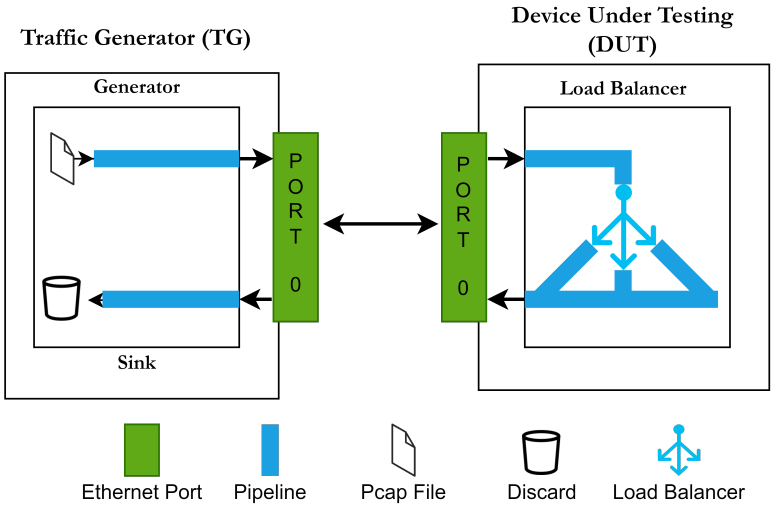


Figure 3.2: Hardware topology Load Balancer

3.2 Traffic Generator

Running on the Traffic Generator (TG) machine is the TG application which is responsible for generating enough traffic to be able to fully saturate the link that interconnects the two machines, as described in Section 3.1. To be able to test the effectiveness of the LB under different scenarios, the TG was developed with the capability to modify certain aspects of its traffic generation. Namely, parameters such as unique destination IP addresses, number of flows and size of packets. By generating traffic with modifications according to these parameters, it is possible to determine how the LB performs under different scenarios.

Specifically, the modification of the 4-tuple (IP addresses and ports) is done by applying a bitmask to each field and ANDing against a pre-defined IP address. Thus, by setting varying degrees of strictness in the bitmask, the number of flows can be controlled. The packet sizes are simply modified by padding with zeroes to the desired length of 1500 bytes.

The TG consists of an element class named `TrafficGenerator` containing the logic for adjusting packet size and flow count. An `AverageCounterIMP`

element is used to measure statistics and lastly, a `ToDPDKDevice` element sends the packets to the underlying I/O layer to be sent to the NIC.

The `TrafficGenerator`, as seen in Fig. 3.4a, element class reads data from trace files with `FromMinDump`, sends it through a chain of `StoreData` which modifies source address, a destination address, and source and destination ports according to a corresponding bitmask for each field. Then the packet is sent to a `Switch` element which as the name implies sends the packet to different places depending on a set value. This setup allows for per-core independent processing of all returning packets. Thus allowing for linear scaling in performance with an increase in core count. Based on the size of the packets the packet is then sent along to one of four destinations depending on what size has been set. After the size has been adjusted the packet exits the element class. `AverageCounterIMP` is used to measure the packet count and packet rate averages. The version we use is a special thread-safe version. `ToDPDKDevice` sends the packet to the chosen port of the NIC set in DPDK at the start of FastClick.

The sink, seen in Fig. 3.4b, is where the packets are returned after they have visited the other machine. This is where statistics like latency and throughput are collected based on round trip time and packet counters.

`FromDPDKDevice` gets the packets from, DPDK and by extension, the NIC and sends them to the next element.

The `AverageCounterIMP` counts the packets again to have values to compare with the ones acquired in the generator for the packets to then be sent along to the `Classifier`.

The `Classifier` in its turn sorts out which packets are supposed to be recorded a timestamp for in the on of the Sinks or only discarded in with the `Discard` element.

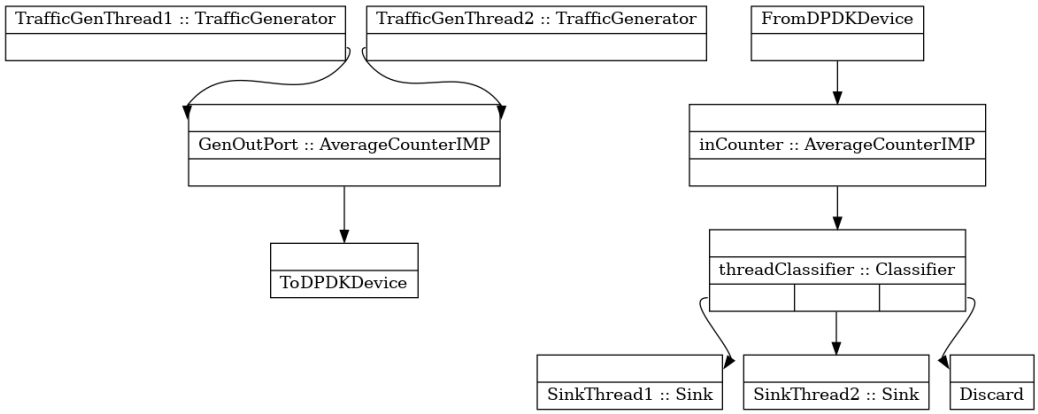
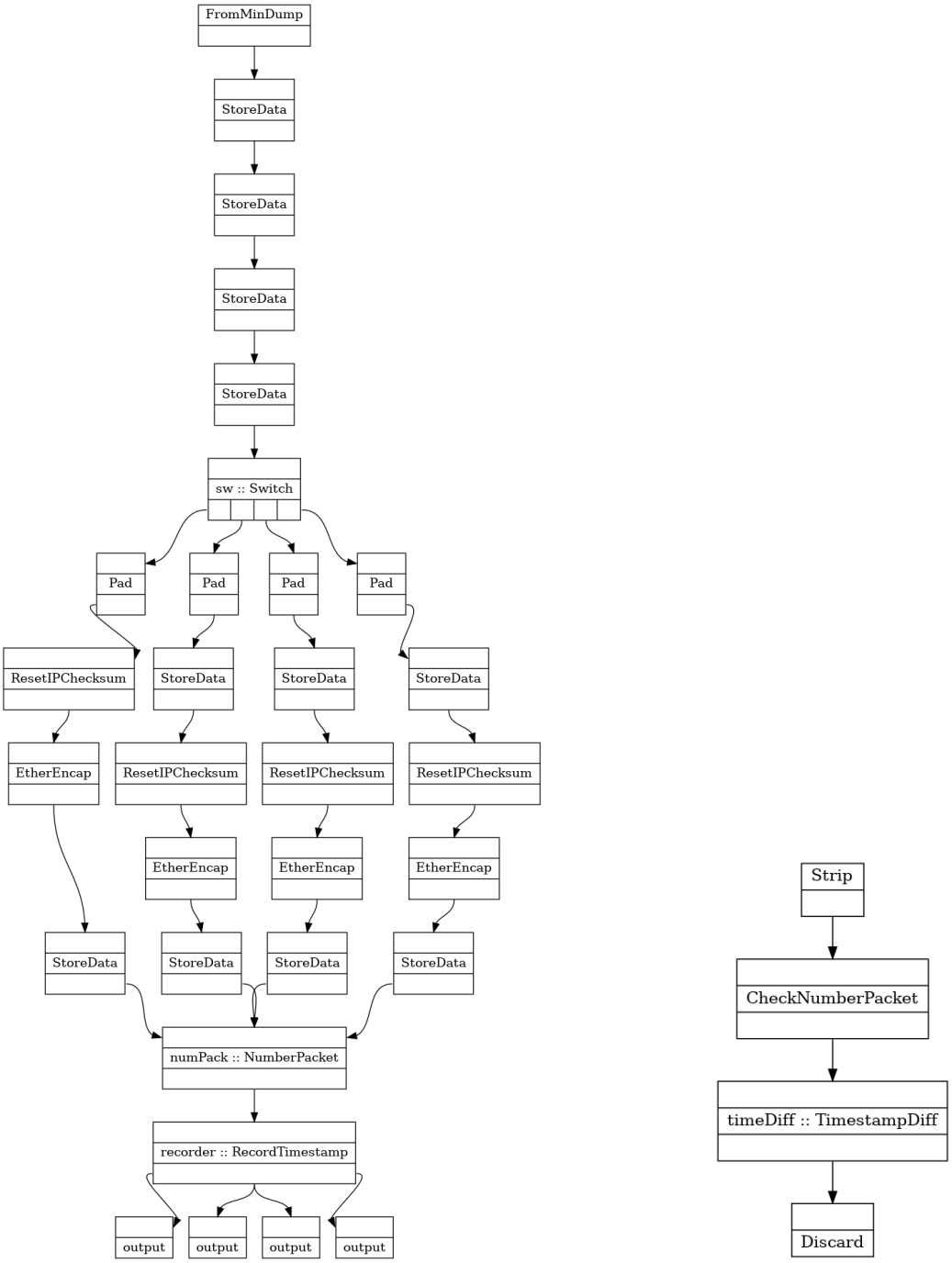


Figure 3.3: Internal structure of the Traffic Generator showing the generator part and the sink part.



(a) Internal structure of the `TrafficGenerator` element

(b) Internal structure of the `sink` element

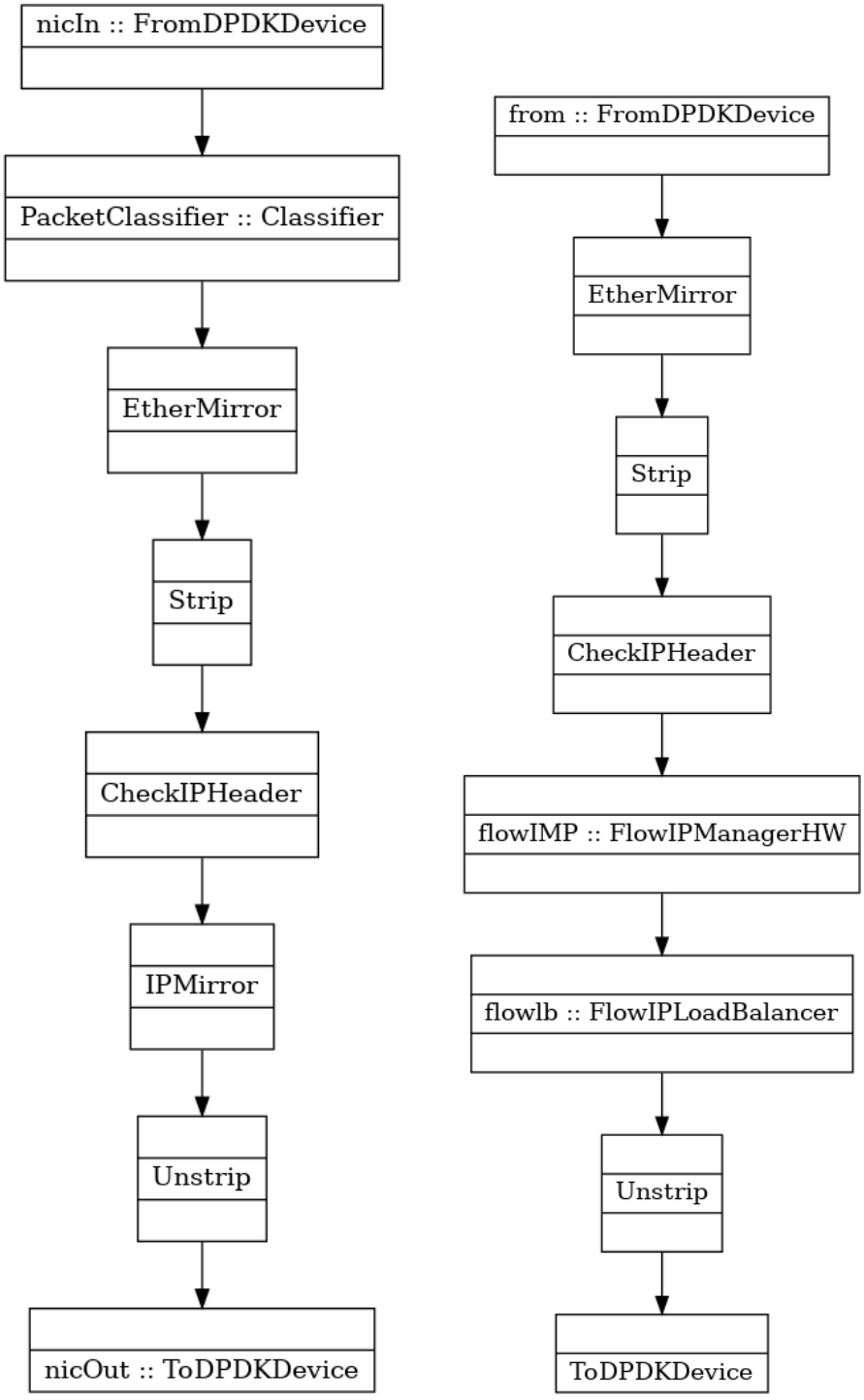
Figure 3.4: Traffic Generator Elements

3.3 Basic Forwarder

An BF was developed, as shown in Fig. 3.1, to evaluate the baseline performance of the testbed with as little server-side computation as possible by relaying the packets received from the TG. Its structure can be found in Fig. 3.5a. It classifies the packets, received from DPDK, and further processes IP-packets only. Then the MAC addresses are swapped in the Ethernet header before stripping the header, to be able to process the IP part of the packet. The IP header is validated, and if it is a valid header, the IP addresses are swapped. Otherwise, the packet is discarded. Then the Ethernet headers are unstripped and it is forwarded to DPDK again.

3.4 Load Balancer

The structure of the LB can be found in Fig. 3.5b. It shares much of the same structure as the BF but relies on two extra elements for load balancing and connection tracking. One element for classifying the packet into a specific flow, and one element to map the specific flow to a destination IP.



(a) Basic Forwarder (b) Load Balancer

Figure 3.5: Internal structure of Basic Forwarder and Load Balancer.

3.4.1 Underlying Data Structures

To be able to classify flows in almost constant time, the `FlowIPManagerHW`, which makes use of a hash table for each CPU core used. Their combined size equals the total capacity of the `FlowIPManagerHW`, which it is initiated with. The hashes for each flow are computed by looking at the 5-tuple, which consists of source and destination IP addresses and ports as well as protocol.

To avoid thread synchronisation when packets try to access `FlowIPManagerHW`, and prevent any processing from being done on more than one thread simultaneously, the table is using core sharding. This sharding as allowed thanks to RSS, available in ”modern” NICs, ensure that each specific flow will always arrive on the same queue. This allows to speed up packet handling, while the capacity to store flows is decreased. All flows have to be registered the first time that they are seen on a specific thread.

3.5 Load Balancers with Hardware Offloading

The LB logic used when offloading to hardware is the same as without offloading to hardware. The difference is in deciding what to offload to hardware. The `FlowIPManagerHW` element is used, which is based on `FlowIPManagerIMP` with modifications. The `FlowIPManagerHW` element can be initialized with and without offloading to hardware, as well as with different offloading techniques.

3.5.1 Hardware Offloading

Offloading to the hardware allow for utilization of full hardware speed. By avoiding the process of software functions, the latency should decrease for the packets belonging to the offloaded flows. However, it is not possible to offload beneficially without specifying what to offload, as presented in section 2.3.

3.5.2 Optimally Offloading Flow Rules

To optimally offload packets would be to decrease the latency as much as possible for all packets on average. As mentioned previously, packets belonging to flows that are offloaded should have lower latency than packets belonging to flows that are not.

The hash tables on the NIC are much more limited in size than what is available in the software [28]. Therefore there exist a need to classify different flows, and decide which of all flows are best suited for offloading if one wants to do so optimally. As the goal is to decrease the latency as much as possible on average, it leads to the conclusion that flows that send more packets should statistically be offloaded with a higher priority than flows with fewer packets.

3.6 Testing

The tests were run by starting the desired application on the DUT and the generator and sink on the TG, using different values for the variables of interest and using multiple runs to compensate for any statistical anomalies. The tests are automatic and allow for batching to reduce the chance of human error. This also allowed for running them without human supervision.

The main metrics of interest were throughput, latency and packet loss as these are good data points to use for comparison of LBs and how they differ for different core counts, hash table sizes, and flow amount. The core counts were chosen by keeping them aligned to the power of two to follow the sizing of the hash table, and underlying memory allocations aligned. The input variables tested can be seen in Table 3.1.

The test suite is composed as a series of bash scripts running on the TG, using *ssh* to start the remote applications on the DUT. The data of the different runs

Core Count	Hash Table Size	Flow Count	Packet Size
1	2E6	2E6	1500 bytes
2	4E6	8E6	
4	8E6		
8	16E6		

Table 3.1: Tested input variables

were merged with a python script. When running the HO tests the script starts the LB with HO and then run the TG two times in a row, referred to as run 1 and run 2 in the result section. This is done to get results both from when the flows are and are not marked from the beginning, and with this result understand the influence of flow insertion operation.

3.6.1 Data Collection

The data collection was done at the TG and saved to files with CSV format. Table 3.2 presents a condensed example of the data. A python script was used to average the data from multiple runs, to get more statistical validity. Eventually, several Gnuplot [36] scripts were used to generate the desired graphs of the collected data, for further analysis and visualisation of results.

T (s)	TxCnt	RxCnt	TxRt	RxRt	PL	L	L 95%	L 99%
6.1562	1247E4	1166E4	9625E7	9571E2	0.06	196.7	226	231

Table 3.2: Example of the structure of collected data. Variables: Time, TxCount, RxCount, TxRate, RxRate, Packet Loss and latency (ns) for total, 95 percentile and 99 percentile.

Chapter 4

Results

This section presents the results that have been obtained concerning the formulated research question in section 1.3, how effective hardware offloading is for connection tracking in LBs. Histograms, boxplots, and line plots are used to illustrate and provide an overview of the data that has been gathered. Additionally, results from the optimal offloading of flow rules are presented in relation the naïve offloading.

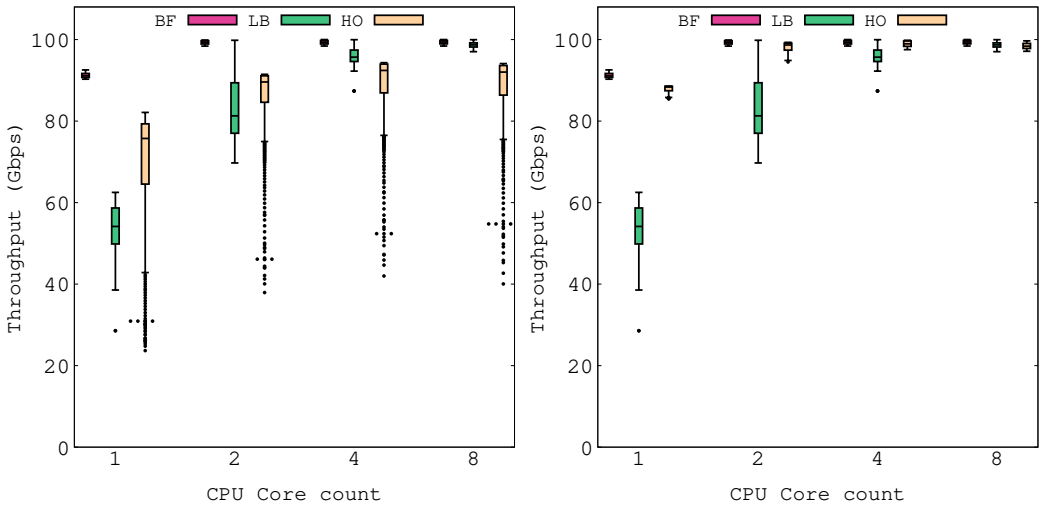
Following are results based on the tests that have been performed, and the data that has been gathered while running the TG against the BF, LB and LB with HO while varying parameters such as the number of flows and hash table capacities. The leftmost graph in each set (run 1), demonstrates the load balancer with hardware offloading running from a cold start. That is, no flows have been seen previously or exist in any tables. The rightmost graphs in each set (run 2), illustrate the subsequent run, where the same trace file is replayed again, but are seen for the second time. They should thus have installed the hash tables and hardware rules respectively, to the degree that the respective capacities allow. Results are then shown for increasing hash table capacity.

Graphs for 2, 4, 8 and 16 million hash table entries exist, some are, however, omitted due to their similarity in results but can be found in Appendix A. The

most significant differences are observed in 2 and 8 million flows, and were thus chosen for the graphs.

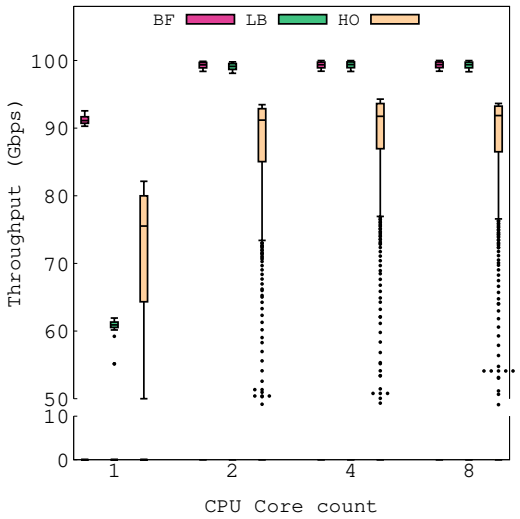
4.1 Throughput

Fig. 4.1 illustrates the difference in throughput of the BF, LB, and LB with naïve HO when balancing 2 million flows as the capacity of the hash tables is set to increase sizes (2 and 8 million). In Fig. 4.1a and Fig. 4.1c the first run done for the LB with HO can be seen, the first run includes accessing the hardware to write flows. While in Fig. 4.1b and Fig. 4.1d the traffic generator is run a second time after all flows to be offloaded have already been so. Therefore they show the throughput when there is no need to wait for additional I/O operations for rule insertions. This test was also done for 4 and 16 million entries, but as these turned out to be close to identical as for 4 million entries they are omitted here.

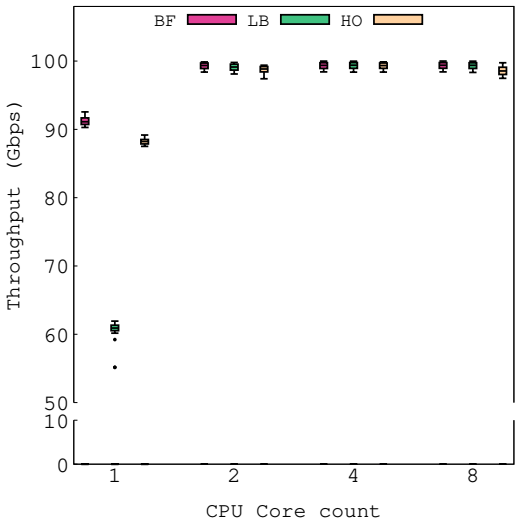


(a) 2M Hash table entries, 2M Flows, Run 1

(b) 2M Hash table entries, 2M Flows, Run 2

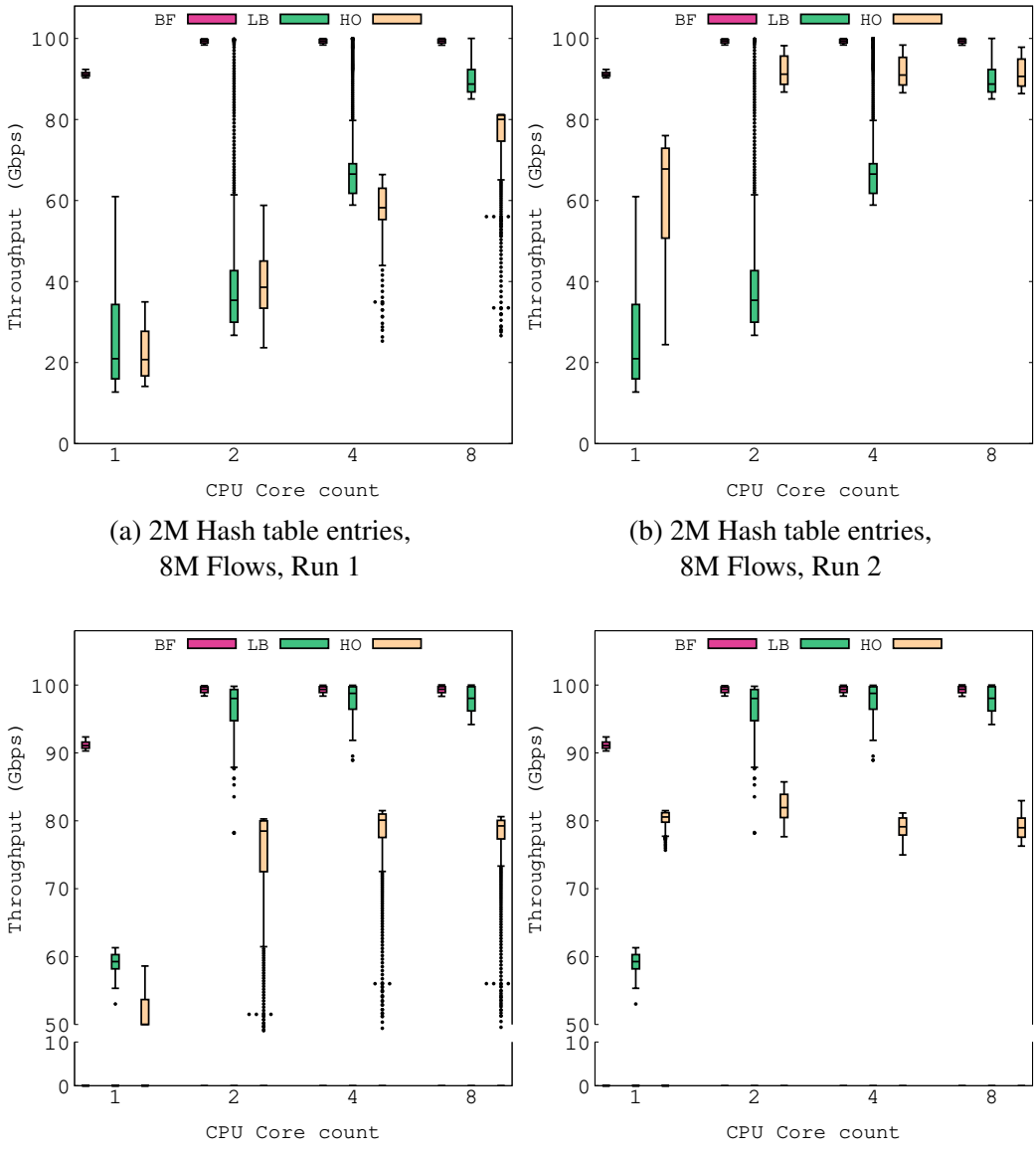


(c) 8M Hash table entries, 2M Flows, Run 1



(d) 8M Hash table entries, 2M Flows, Run 2

Figure 4.1: Throughput versus CPU core count



(a) 2M Hash table entries, 8M Flows, Run 1

(b) 2M Hash table entries, 8M Flows, Run 2

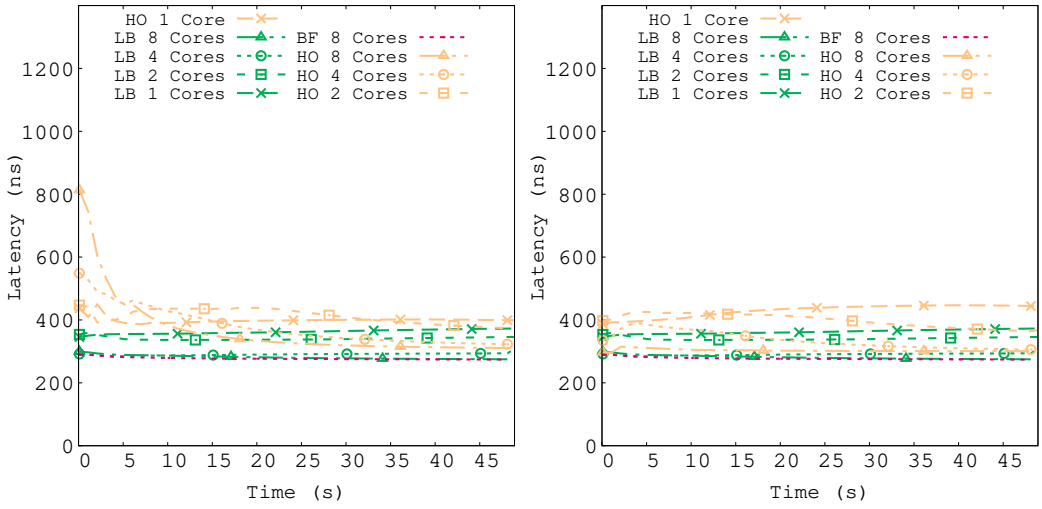
(c) 8M Hash table entries, 8M Flows, Run 1

(d) 8M Hash table entries, 8M Flows, Run 2

Figure 4.2: Throughput versus CPU core count

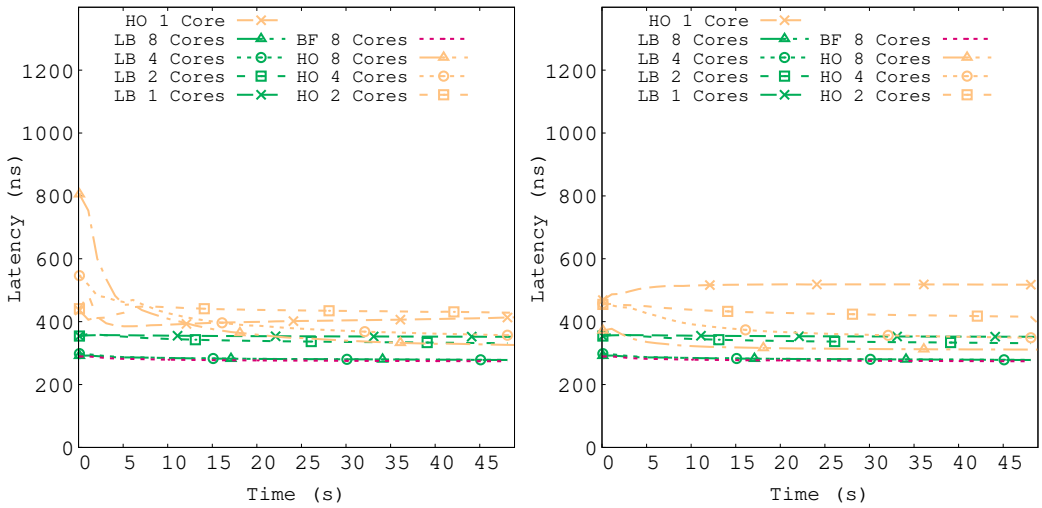
Fig. 4.2 shows the same measurements as Fig. 4.1, but with a table size of four and eight million, respectively, together with eight million flows. Here is shown the difference between when all flows fit into the hash table of the LB and the software part of the LB with HO, versus when they do not.

4.2 Latency



(a) 2M Hash table entries, 2M Flows, Run 1

(b) 2M Hash table entries, 2M Flows, Run 2

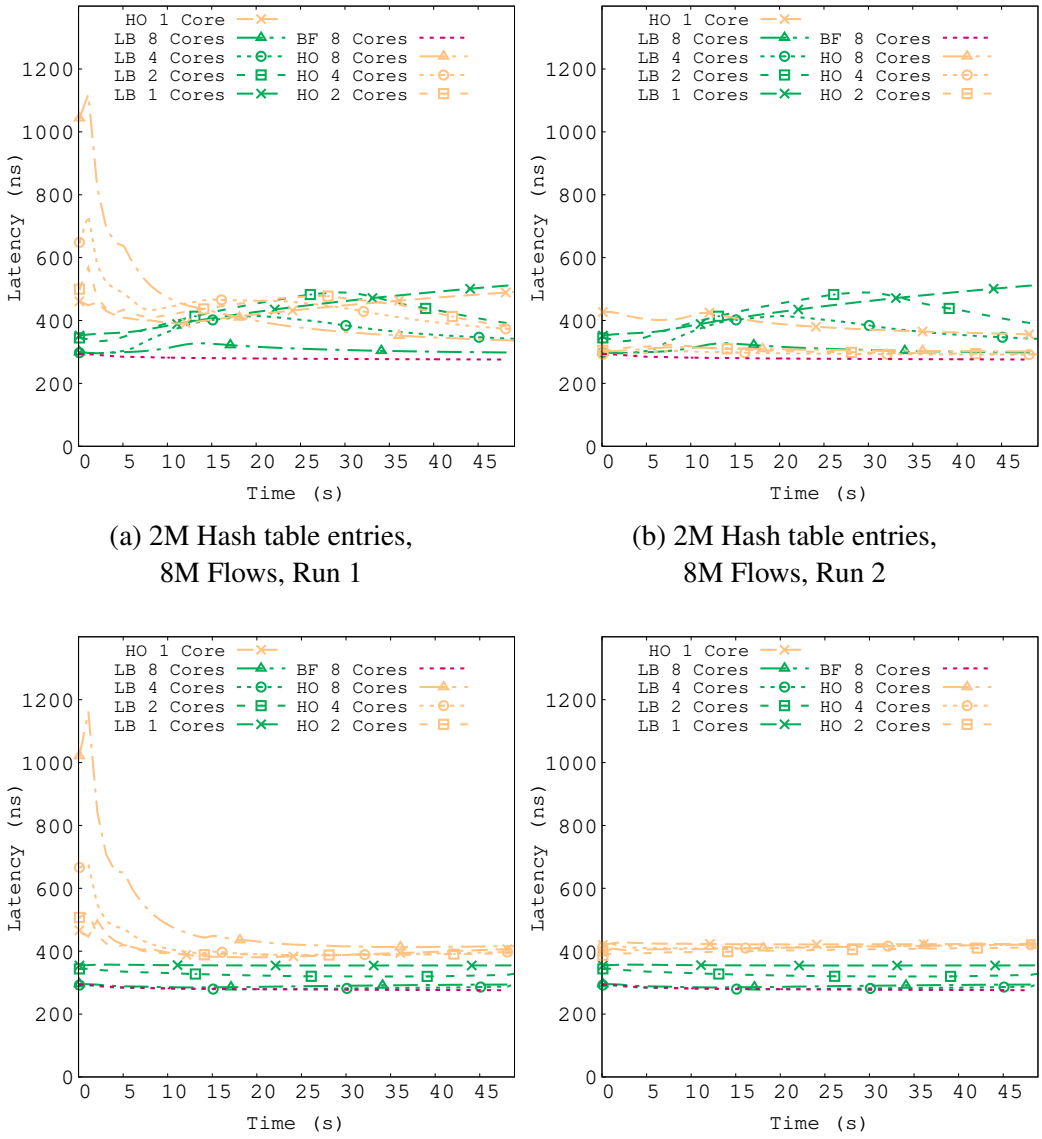


(c) 8M Hash table entries, 2M Flows, Run 1

(d) 8M Hash table entries, 2M Flows, Run 2

Figure 4.3: Latency for different CPU core counts.

In Fig. 4.3 the average packet latency can be seen over time for the BF, LB, and LB with HO depending on number of CPU cores used. Both LB are using a hash table of 2M in Fig. 4.3a and 4.3b. While in Fig. 4.3c and 4.3d a table size of 8M is used. In Fig. 4.3a and 4.3c, rules are inserted into the hardware while running. While in Fig. 4.3b and 4.3d all rules are already inserted.



(a) 2M Hash table entries, 8M Flows, Run 1

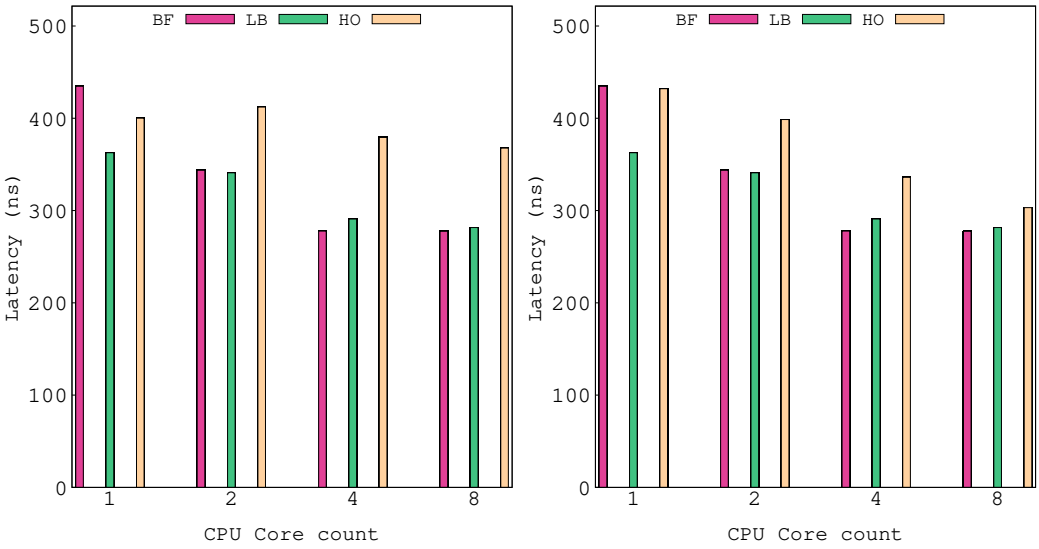
(b) 2M Hash table entries, 8M Flows, Run 2

(c) 8M Hash table entries, 8M Flows, Run 1

(d) 8M Hash table entries, 8M Flows, Run 2

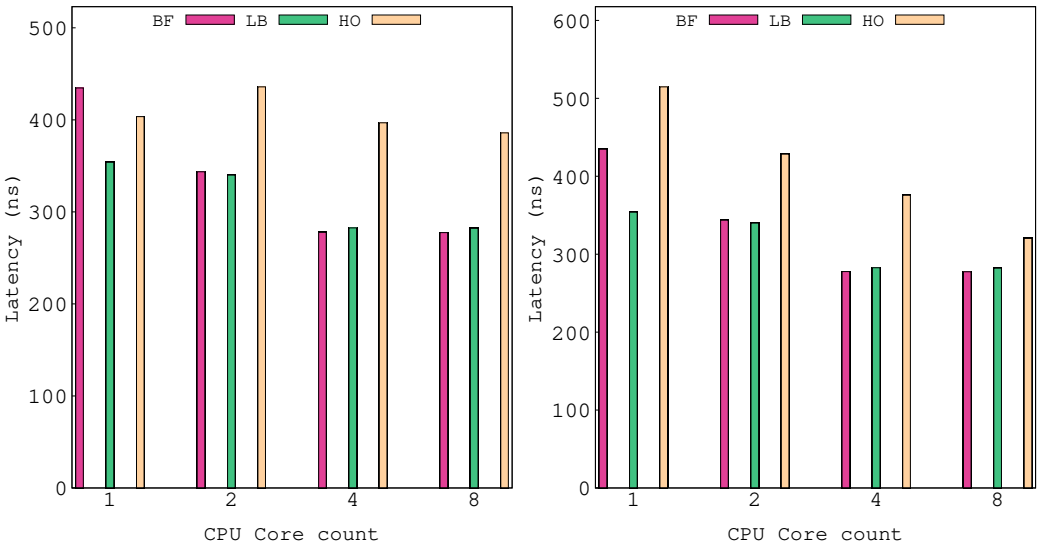
Figure 4.4: Latency for different CPU core counts.

In Fig. 4.4 the same measurements done as for Fig. 4.3, but with 8M flows instead of 2M flows.



(a) 2M Hash table entries, 2M Flows, Run 1

(b) 2M Hash table entries, 2M Flows, Run 2

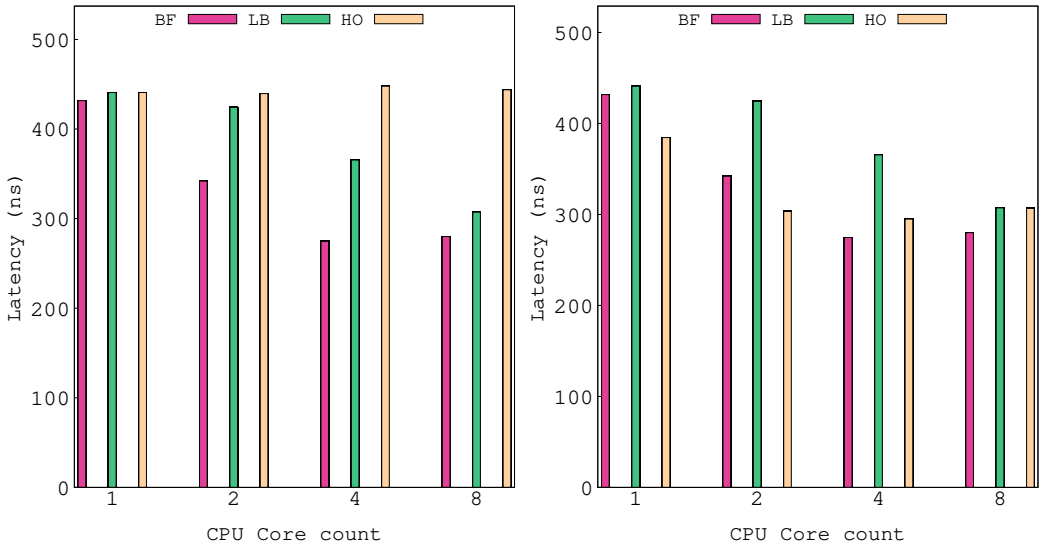


(c) 8M Hash table entries, 2M Flows, Run 1

(d) 8M Hash table entries, 2M Flows, Run 2

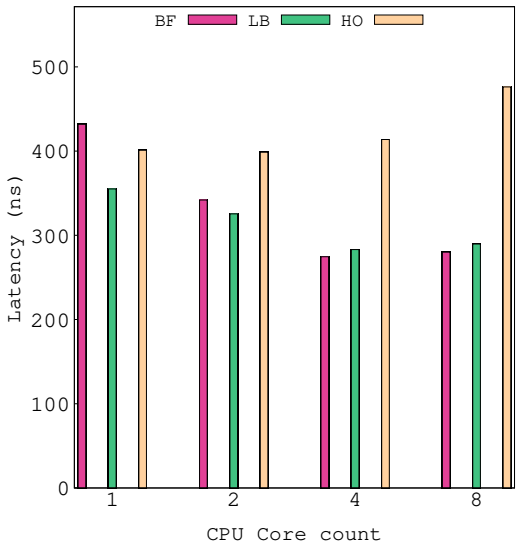
Figure 4.5: Latency versus CPU core counts.

In Fig. 4.5 the same information as could be seen in Fig. 4.3 can be seen, but averaged over the whole run time instead of for each time step.

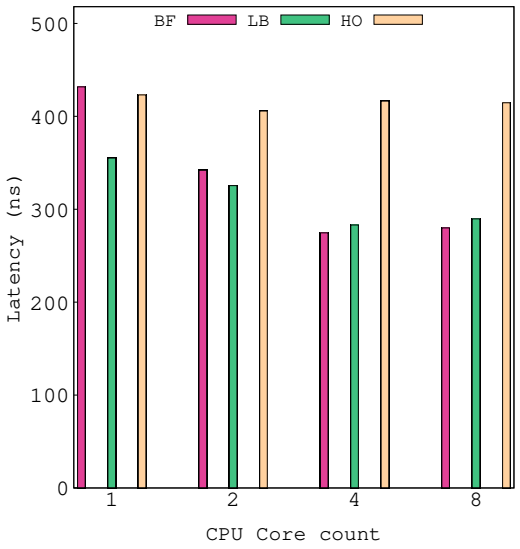


(a) 2M Hash table entries, 8M Flows, Run 1

(b) 2M Hash table entries, 8M Flows, Run 2



(c) 8M Hash table entries, 8M Flows, Run 1



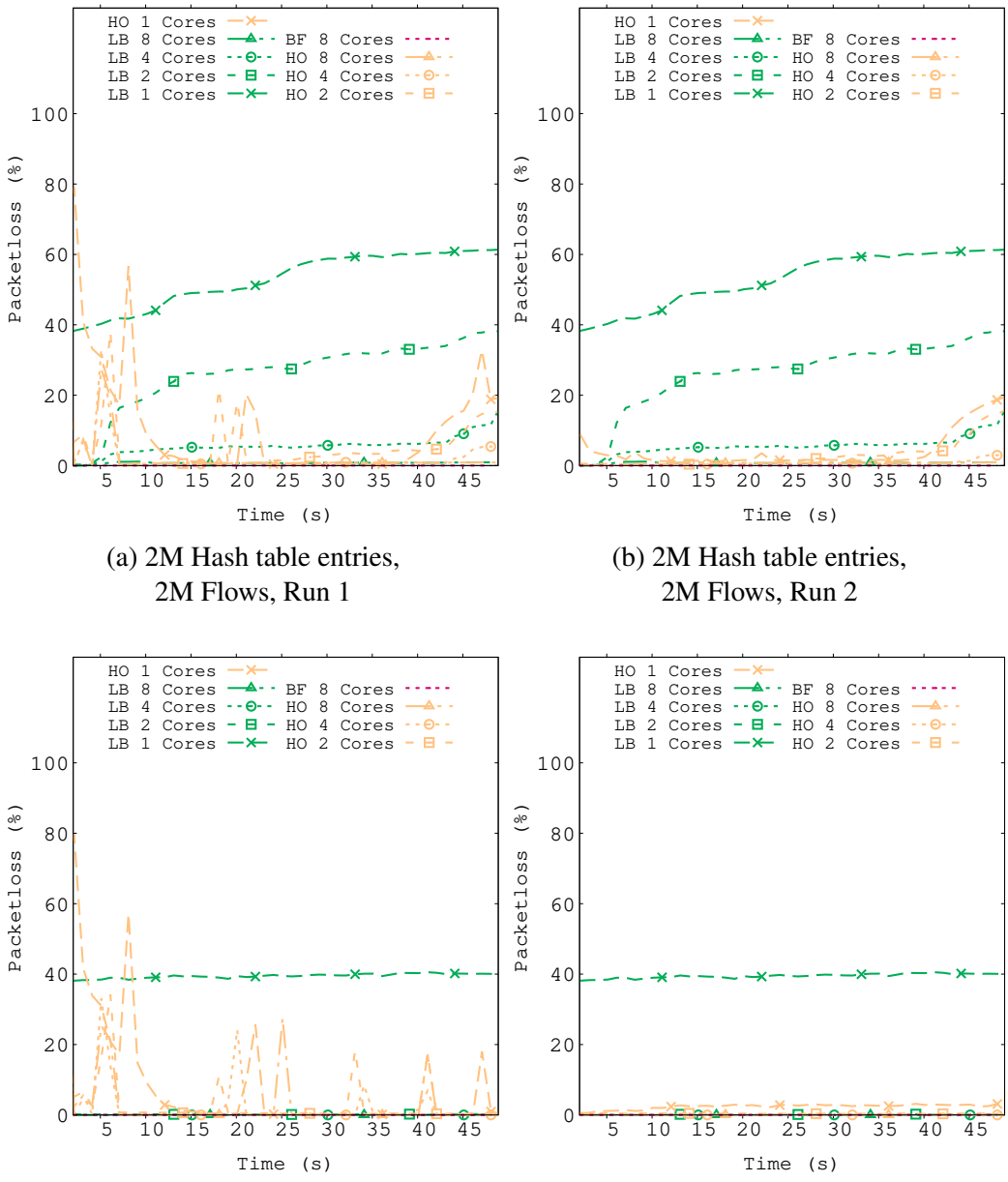
(d) 8M Hash table entries, 8M Flows, Run 2

Figure 4.6: Latency versus CPU core counts.

In Fig. 4.6 as in Fig. 4.5 the average latency over the whole run time is shown. The corresponding latency measurements for each time step can be seen in Fig. 4.4.

4.3 Packet Loss

Following are the results of packet loss presented.



(a) 2M Hash table entries, 2M Flows, Run 1

(b) 2M Hash table entries, 2M Flows, Run 2

(c) 8M Hash table entries, 2M Flows, Run 1

(d) 8M Hash table entries, 2M Flows, Run 2

Figure 4.7: Packet loss for different CPU core counts.

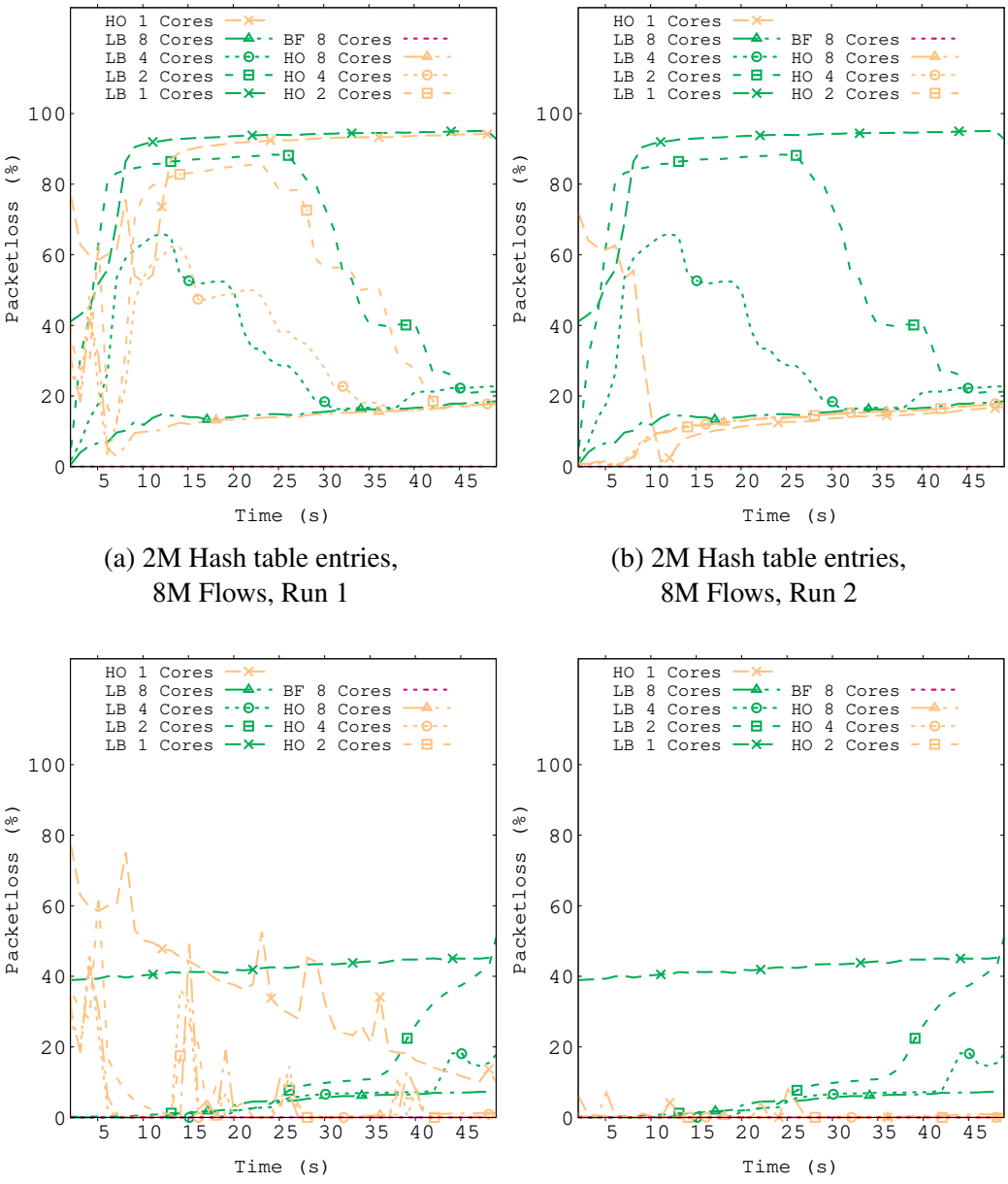
Furthermore, the packet loss for the BF, LB, and LB with HO respectively can be seen for different core configurations when the traffic consists of 2 million flows in Fig. 4.7. The first run when using a hash table with 2 million entries can be seen in Fig. 4.7a, while the second run can be seen in Fig. 4.7b. The same data when using a hash table size of 8 million can be seen in Fig. 4.7c and Fig. 4.7d.

In Fig. 4.8, the same data measurements as Fig. 4.7 can be seen with 8 million flows instead of 2 million flows.

4.4 Optimal Offloading

This section presents the same categories of results as the previous, but with the attempt of making an optimal selection of flows to offload. In this case, based on total flow size.

Fig. 4.9 demonstrates the difference in throughput sustained with the naïve offloading in comparison to offloading with selection of flows depending on their size per amount of core. Similarly, Fig. 4.10 illustrates the average latency between the two methods.



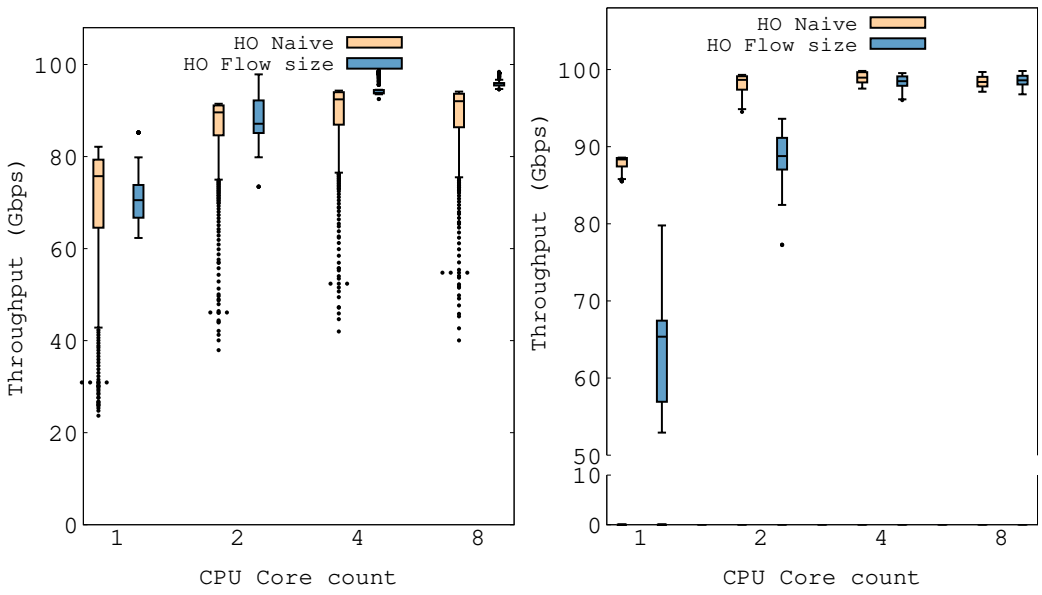
(a) 2M Hash table entries, 8M Flows, Run 1

(b) 2M Hash table entries, 8M Flows, Run 2

(c) 8M Hash table entries, 8M Flows, Run 1

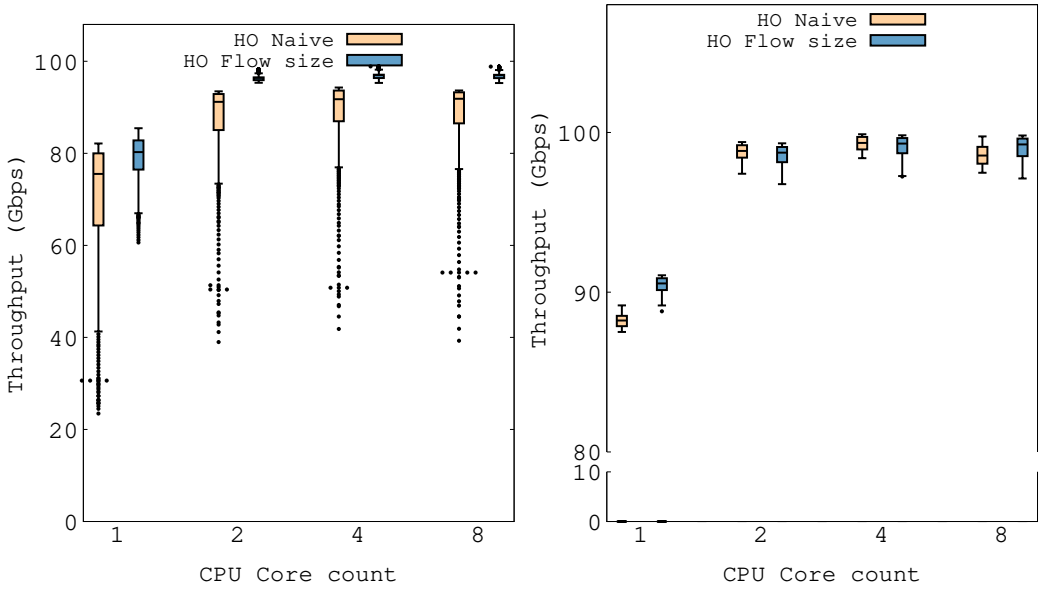
(d) 8M Hash table entries, 8M Flows, Run 2

Figure 4.8: Packet loss for different CPU core counts.



(a) 2M Hash table entries, 2M Flows, Run 1

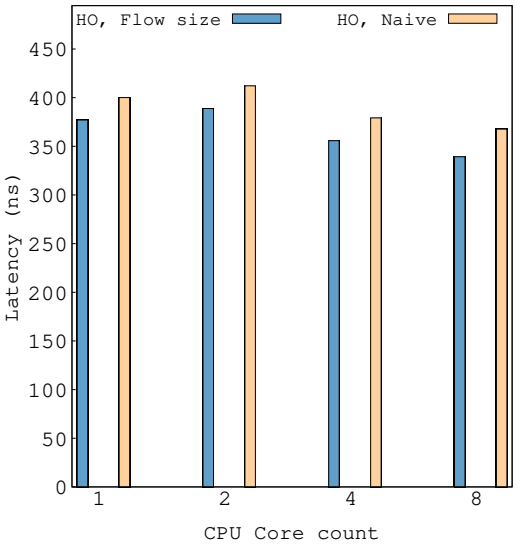
(b) 2M Hash table entries, 2M Flows, Run 2



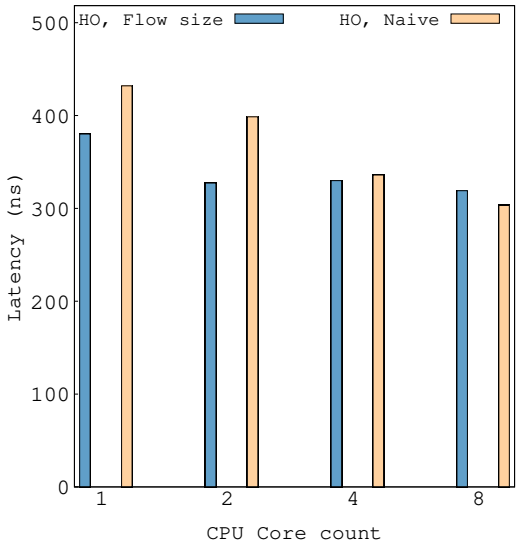
(c) 8M Hash table entries, 2M Flows, Run 1

(d) 8M Hash table entries, 2M Flows, Run 2

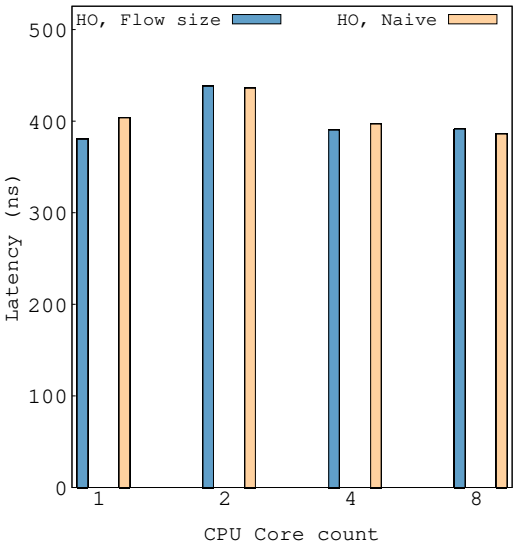
Figure 4.9: Throughput versus CPU core count.



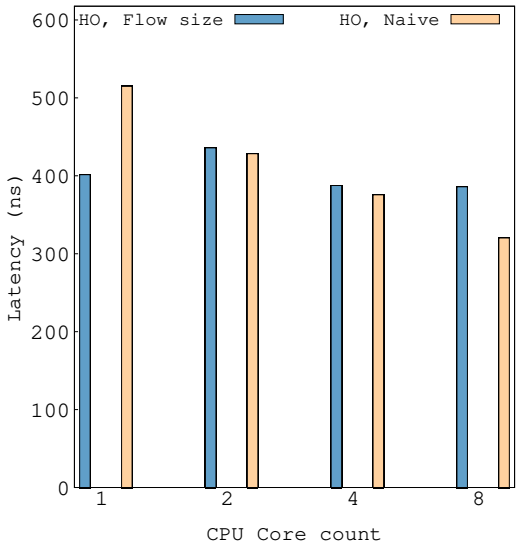
(a) 2M Hash table entries, 2M Flows, Run 1



(b) 2M Hash table entries, 2M Flows, Run 2



(c) 8M Hash table entries, 2M Flows, Run 1



(d) 8M Hash table entries, 2M Flows, Run 2

Figure 4.10: Average latency versus CPU core count.

Chapter 5

Analysis

This section describes what can be interpreted by the presented results in Section 4. First, the throughput is discussed, followed by the packet loss, then latency, and lastly the difference between flow classification techniques.

5.1 Throughput

From Fig. 4.1 and Fig. 4.2 it can be seen, as expected, that the BF has the highest throughput with the lowest variance. This is because when running the BF there are no expensive logical operations being performed at the receiving side. However, when the LB and LB with HO are introduced with their limited hash table sizes, it can be observed that the throughput is affected. This can especially be seen as the underlying tables become overfilled, as well as when the flows are being inserted. The variance for the LB with HO is large during run one, mainly due to the fact that the throughput is low at the start when the rules have to be inserted. However, as rules are inserted and flows are recognized, the throughput starts to increase again as the impact of inserting rules starts to diminish. A fact that is clearly illustrated during run two, where no rules are created at all - and

variance is small and throughput at its best. Overall, it can be observed that when the table size is too small for the amount of incoming flows, all LB configurations suffer performance losses. Additionally, that inserting rules into hardware is a costly operation.

5.2 Packet Loss

From Fig. 4.7 and 4.8 it is seen that when offloading the classification of some flows to the hardware, a lower packet loss can be achieved than when all classification is done in software. The difference is most noticeable when using one CPU core, and a hash table too small to contain all the flows. As additional cores are used, the difference in the number of packets lost between the two, naïve and selective offloading, diminishes.

5.3 Latency

The graphs for 2 million and 8 million hash table entries were chosen, as the larger table sizes showed no relevant difference in results. In Fig. 4.3c it can be seen that the latency is overall slightly higher than in Fig. 4.3a. While the same is seen in Fig. 4.3d and Fig. 4.3b the difference is easier to see there. It is clear that the difference in latency between using the LB with or without HO becomes smaller with the increase in the number of cores.

5.4 Optimally Offloading Flow Rules

The performance comparison between LBs with HO in Fig. 4.9 for throughput shows that when selectively applying offloading to flows depending on how many

bytes have been seen belonging to that specific flow, the performance is not affected as much in the first run as seen in Fig. 4.9a and 4.9c as when naively offloading every flow seen in order. Furthermore, in Fig. 4.9b and 4.9d it can be seen that the performance after flow insertion as well does not differ in any great amount between selectively offloading flows, or offloading naively.

In Fig. 4.10 it can be seen that on average, there is not a big difference between the latency for packets between selectively or naively offloading flows. Though in some specific situations a difference can be observed. Especially lower core counts seem to favour selective offloading, both on the first and second run, while higher core counts seem to allow lower latency for the naïve solution.

5.5 Causes of Error

Some potential error margins in the execution of the study, and how these could be eliminated, are presented here. The traffic relayed by the TG is, as stated in section 1.5, limited to IP packets. It would have been desirable to run the test with different types of traffic. However, since most traffic in the real world consists of IP packets, this is not considered a margin of error that significantly affects the results.

Sometimes the latency is measured as negative, this could be due to discrepancies in the placement of the latency counter in the TG on the sending and receiving side. Also, packets sometimes return faster than expected which together with the small time frame that packets are returned in, gives unexpected values in latency measurements.

It was also occasionally observed that the received packets count was higher than the sent count, this is assumed to be due to the batching used in FastClick for handling packets. The batching mode holds packets until a certain amount

of packets have been handled, and thus more packets could be returned during a specific time window than the number of packets sent during the same. This could lead to negative packet loss and latency measurements as timestamps are stored within the packets.

Furthermore, the authors have noticed a discrepancy in the results seen in Fig. 4.1 for the hardware-assisted load balancing versus the normal load balancing. The expected outcome is an increased performance with a larger hash table size, but as observed the performance decreases in Fig. 4.2b in comparison to Fig. 4.2d. As multiple runs have been performed, and the results averaged, the cause should not be due to any run-specific anomalies. An explanation for this could instead be ineffective hardware offloading, i.e. with such a large data structure there are a lot of cache misses. However, more testing would be required to confirm this.

Chapter 6

Conclusions

The main outcome of the study shows that offloading the connection tracking to hardware has its performance gains in the long term. However, inserting flows into the hardware is a costly operation that has an initial negative impact. If flows are selectively offloaded, this impact can be mitigated to some extent, as not all flows are inserted at once. Initial results also indicate that the throughput remains more stable over time. Finally, there seems to be some extra performance that can be gained when doing the selective offloading, but further investigation into the offloading threshold parameter, as well as the selection technique, is required to maximize the performance gains.

To summarize, if the incoming flows arrive at a reasonable rate, hardware offloading seems to be effective at handling the connection tracking and freeing up performance for the CPU. However, if the flows are arriving at a high intensity or are too short-lived, the impact of the hardware offloading will have detrimental effects. If flows are long-lived, it can be argued that the initial negative impact is negligible in comparison to the performance gained over time. Depending on the underlying application, these consequences may have different impact. For example, users browsing a website where connections are short lived and

small would likely see an increase in latency in the naïve case. On the contrary, long lived flows consisting large packets, such as in streaming applications or downloads, would suffer less.

If the underlying technology for hardware offloading and the offloading technique continues to evolve, HO could likely have a great impact on the performance and power consumption in connection tracking scenarios leading to lower costs and better scalability.

6.1 Future work

Due to the project being limited in time and scope, many parameters of interest have been ignored. Parameters that could potentially yield additional performance gains. Most notably, this project only tested a single selection technique in the attempt to optimally offload flows, but there are many different characteristics of the incoming flows that could be considered. For example, how long flows last, packet count and size. This study only focused on the latter, and the offloading threshold was set to what seemed a reasonable value. However, more performance benefits could likely be seen if this parameter was chosen with more care and additional traces and real application data utilized. Additionally, flow table maintenance algorithms used to remove the ”expired” flows have not been tested, further investigation into these algorithms could be made.

The largest impact on performance was when new flows were inserted into hardware. If improvements could be made to the insertion time into hardware, the offloading would prove even more beneficial. It would also allow the hardware offloading to better handle very dynamic traffic with many short and long-lived flows.

Furthermore, this project used a single set of trace files for its traffic generator.

While the traffic is captured from public routers where traffic, most likely, varies - different results could be seen if other traces were used.

Bibliography

[1] Cisco. (). "Cisco annual internet report - cisco annual internet report (2018–2023) white paper", Cisco, [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html (visited on 12/21/2021).

[2] K. Routh and T. Pal, "A survey on technological, business and societal aspects of internet of things by q3, 2017", in *2018 3rd International Conference On Internet of Things: Smart Innovation and Usages (IoT-SIU)*, Bhimtal: IEEE, Feb. 2018, pp. 1–4, ISBN: 9781509067855. DOI: 10.1109/IoT-SIU.2018.8519898. [Online]. Available: https://ieeexplore.ieee.org/document/8519898/ (visited on 12/22/2021).

[3] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling", *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3, pp. 365–376, Jun. 22, 2011, ISSN: 0163-5964. DOI: 10.1145/2024723.2000108. [Online]. Available: https://dl.acm.org/doi/10.1145/2024723.2000108 (visited on 11/09/2021).

[4] P. E. McKenney, "Is parallel programming hard, and, if so, what can you do about it? (second edition)", *arXiv:1701.00854 [cs]*, Mar. 31, 2021. arXiv: 1701.00854. [Online]. Available: http://arxiv.org/abs/1701.00854 (visited on 11/09/2021).

[5] A. Håkansson, "Portal of research methods and methodologies for research projects and degree projects", presented at the The 2013 World Congress in Computer Science, Computer Engineering, and Applied Computing WORLDCOMP 2013; Las Vegas, Nevada, USA, 22-25 July, CSREA Press U.S.A, 2013, pp. 67–73. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-136960 (visited on 11/09/2021).

[6] (). "CAIDA data - overview of datasets, monitors, and reports", CAIDA, [Online]. Available: https://www.caida.org/catalog/datasets/overview/ (visited on 12/21/2021).

[7] G. Kamiya. (Jun. 2020). "Tracking data centres and data transmission networks 2020 – analysis", IEA, [Online]. Available: https://www.iea.org/reports/tracking-data-centres-and-data-transmission-networks-2020 (visited on 12/21/2021).

[8] United Nations. (Dec. 21, 2021). "THE 17 GOALS — sustainable development", THE 17 GOALS, [Online]. Available: https://sdgs.un.org/goals (visited on 12/21/2021).

[9] Sverige and Naturvårdsverket, *Recycling and disposal of electronic waste: health hazards and environmental impacts.* Stockholm: Naturvårdsverket, 2011, OCLC: 939021028, ISBN: 978-91-620-6417-4.

[10] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein,

"Maglev: A fast and reliable software network load balancer", presented at the 13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16), 2016, pp. 523–535, ISBN: 978-1-931971-29-4. [Online]. Available: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eisenbud (visited on 11/09/2021).

[11] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "RouteBricks: Exploiting parallelism to scale software routers", in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09*, Big Sky, Montana, USA: ACM Press, 2009, p. 15, ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629578. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1629575.1629578 (visited on 11/09/2021).

[12] R. Durner, A. Varasteh, M. Stephan, C. M. Machuca, and W. Kellerer, "HNLB: Utilizing hardware matching capabilities of NICs for offloading stateful load balancers", in *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, Shanghai, China: IEEE, May 2019, pp. 1–7, ISBN: 978-1-5386-8088-9. DOI: 10.1109/ICC.2019.8761434. [Online]. Available: https://ieeexplore.ieee.org/document/8761434/ (visited on 11/09/2021).

[13] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang, "Duet: Cloud scale load balancing with hardware and software", *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 27–38, Aug. 17, 2014, ISSN: 0146-4833. DOI: 10.1145/2740070.2626317. [Online]. Available: http://doi.org/10.1145/2740070.2626317 (visited on 11/09/2021).

[14] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen, "Raising the bar for using GPUs in software packet processing", presented at the 12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15), 2015, pp. 409–423, ISBN: 978-1-931971-21-8. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/kalia (visited on 11/09/2021).

[15] Nvidia, *ConnectX-6 SmartNic datasheet*. [Online]. Available: https://nvdam.widen.net/s/q7jxvtktrf/connectx-6-en-smartnic-datasheet-1730950 (visited on 12/22/2021).

[16] I. Corporation, *Intel® ethernet network adapter e810 product brief*, Version 5. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/210969/intel-ethernet-network-adapter-e8102cqda2.html.

[17] Marvell®, *Marvell® LiquidIO™ III*, Sep. 2020. [Online]. Available: https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-liquidio-III-solutions-brief.pdf (visited on 12/22/2021).

[18] T. Barbette, C. Tang, H. Yao, D. Kostić, G. Q. M. Jr, P. Papadimitratos, and M. Chiesa, "A high-speed load-balancer design with guaranteed per-connection-consistency", presented at the 17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20), 2020, pp. 667–683, ISBN: 978-1-939133-13-7. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/barbette (visited on 11/09/2021).

[19] P. R. 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)*, ISSN: 2325-5927, Dec. 2010, pp. 210–213. DOI: 10.1109/TISC.2010.5714641.

[20] J. Xu and M. Singhal, "Cost-effective flow table designs for high-speed routers: Architecture and performance evaluation", *IEEE Transactions on Computers*, vol. 51, no. 9, pp. 1089–1099, Sep. 2002, ISSN: 1557-9956. DOI: 10.1109/TC.2002.1032627.

[21] (Nov. 9, 2021). "DPDK documentation — data plane development kit 21.11.0-rc2 documentation", [Online]. Available: https://doc.dpdk.org/guides/ (visited on 11/09/2021).

[22] C. Dovrolis, B. Thayer, and P. Ramanathan, "HIP: Hybrid interrupt-polling for the network interface", *ACM SIGOPS Operating Systems Review*, vol. 35, no. 4, pp. 50–60, Oct. 2001, ISSN: 0163-5980. DOI: 10.1145/506084.506089. [Online]. Available: https://dl.acm.org/doi/10.1145/506084.506089 (visited on 11/09/2021).

[23] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing", in *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. [Online]. Available: http://ieeexplore.ieee.org/document/7110116/ (visited on 11/09/2021).

[24] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, "The click modular router", in *Proceedings of the seventeenth ACM symposium on Operating systems principles - SOSP '99*, Charleston, South Carolina, United States: ACM Press, 1999, pp. 217–231, ISBN: 978-1-58113-140-6. DOI: 10.1145/319151.319166. [Online]. Available: http://portal.

acm.org/citation.cfm?doid=319151.319166 (visited on 11/09/2021).

[25] A. Farshin, T. Barbette, A. Roozbeh, G. Q. Maguire Jr., and D. Kostić, "PacketMill: Toward per-core 100-gbps networking", in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Virtual USA: ACM, Apr. 19, 2021, pp. 1–17, ISBN: 9781450383172. DOI: 10.1145/3445814.3446724. [Online]. Available: https://dl.acm.org/doi/10.1145/3445814.3446724 (visited on 12/22/2021).

[26] J. Rajahalme, A. Conta, B. Carpenter, and S. Deering. (Mar. 2004). "IPv6 flow label specification", [Online]. Available: https://datatracker.ietf.org/doc/html/rfc3697 (visited on 12/17/2021).

[27] N. Brownlee, C. Mills, and G. Ruth. (Oct. 1999). "Traffic flow measurement: Architecture", [Online]. Available: https://www.ietf.org/rfc/rfc2722.txt (visited on 12/17/2021).

[28] G. P. Katsikas, T. Barbette, M. Chiesa, D. Kostic, and G. Q. Maguire Jr., "What you need to know about (smart) network interface cards", presented at the Passive and Active Measurement - 22nd International Conference, PAM 2021, Virtual Event, March 29 - April 1, 2021, Springer Nature, 2021. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-292353 (visited on 12/23/2021).

[29] Liang Guo and I. Matta, "The war between mice and elephants", in *Proceedings Ninth International Conference on Network Protocols. ICNP 2001*, Riverside, CA, USA: IEEE Comput. Soc, 2001, pp. 180–188, ISBN: 9780769514291. DOI: 10.1109/ICNP.2001.992898. [Online].

Available: http://ieeexplore.ieee.org/document/992898/ (visited on 12/15/2021).

[30] W. Liu, W. Qu, Z. Liu, K. Li, and J. Gong, "Identifying elephant flows using a reversible MultiLayer hashed counting bloom filter", in *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*, Jun. 2012, pp. 246–253. DOI: 10.1109/HPCC.2012.41.

[31] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri, "Ananta: Cloud scale load balancing", in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, ser. SIGCOMM '13, New York, NY, USA: Association for Computing Machinery, Aug. 27, 2013, pp. 207–218, ISBN: 978-1-4503-2056-6. DOI: 10.1145/2486001.2486026. [Online]. Available: http://doi.org/10.1145/2486001.2486026 (visited on 11/09/2021).

[32] T. Cui, W. Zhang, K. Zhang, and A. Krishnamurthy, "Offloading load balancers onto SmartNICs", in *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, Hong Kong China: ACM, Aug. 24, 2021, pp. 56–62, ISBN: 978-1-4503-8698-2. DOI: 10.1145/3476886.3477505. [Online]. Available: https://dl.acm.org/doi/10.1145/3476886.3477505 (visited on 11/09/2021).

[33] R. Durner and W. Kellerer, "Network function offloading through classification of elephant flows", *IEEE Transactions on Network and Service Management*, vol. 17, no. 2, pp. 807–820, Jun. 2020, ISSN: 1932-4537. DOI: 10.1109/TNSM.2020.2976838.

[34] J. R. Quinlan, *C4.5: programs for machine learning*, ser. The Morgan Kaufmann series in machine learning. San Mateo, Calif: Morgan Kaufmann Publishers, 1993, 302 pp., ISBN: 978-1-55860-238-0.

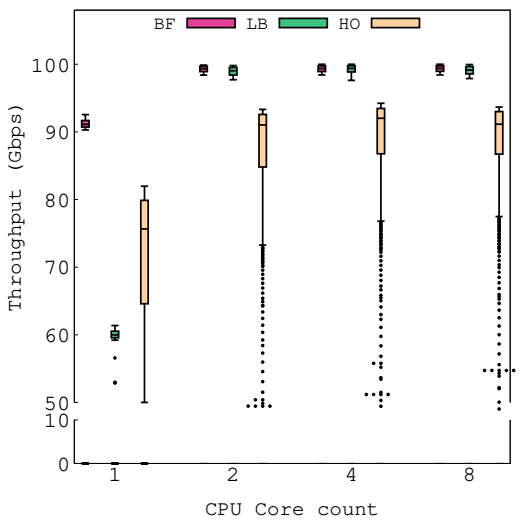
[35] Nvidia, *ConnectX-5 SmartNic datasheet*. [Online]. Available: https://www.mellanox.com/products/ethernet-adapter-ic/connectx-5-en-ic (visited on 12/22/2021).

[36] (). "Gnuplot homepage", [Online]. Available: http://www.gnuplot.info/ (visited on 11/09/2021).

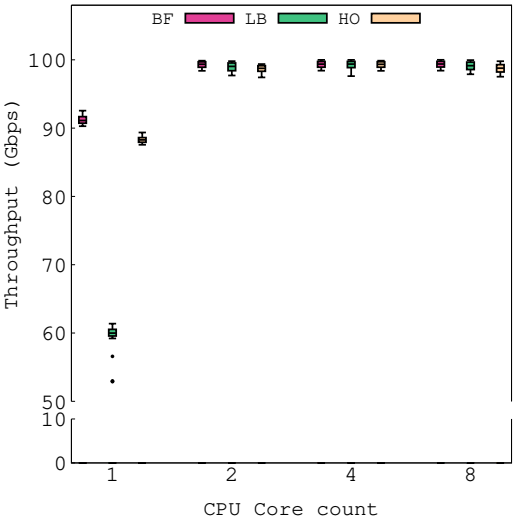
Appendix A

Graphs

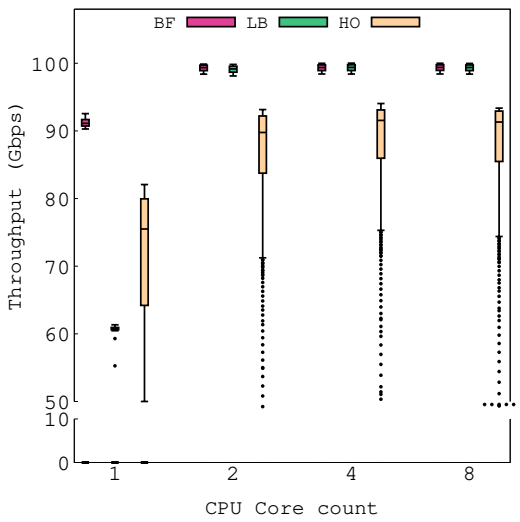
In this appendix we have collected all unused graphs for the ones curious to see.



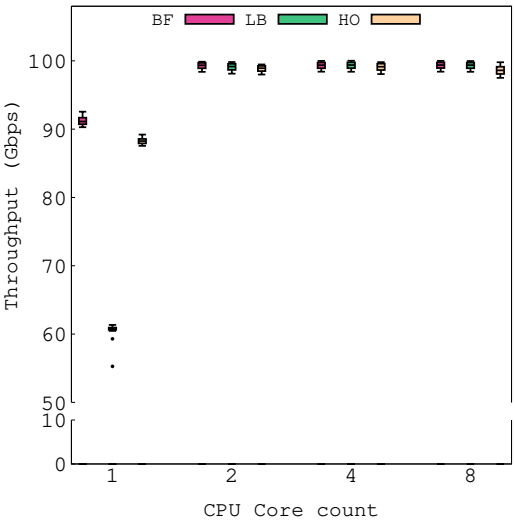
(a) 4M Hash table entries, 2M Flows, Run 1



(b) 4M Hash table entries, 2M Flows, Run 2

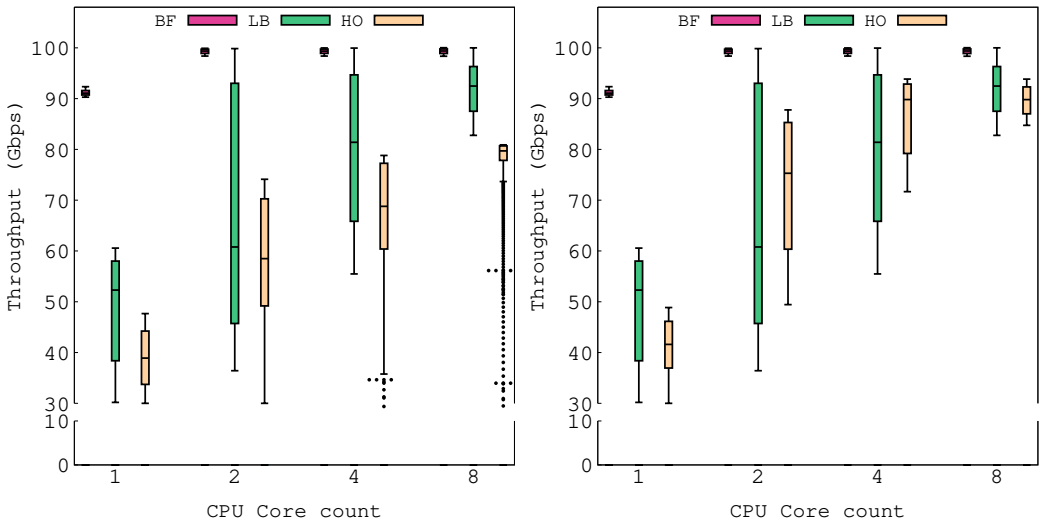


(c) 16M Hash table entries, 2M Flows, Run 1



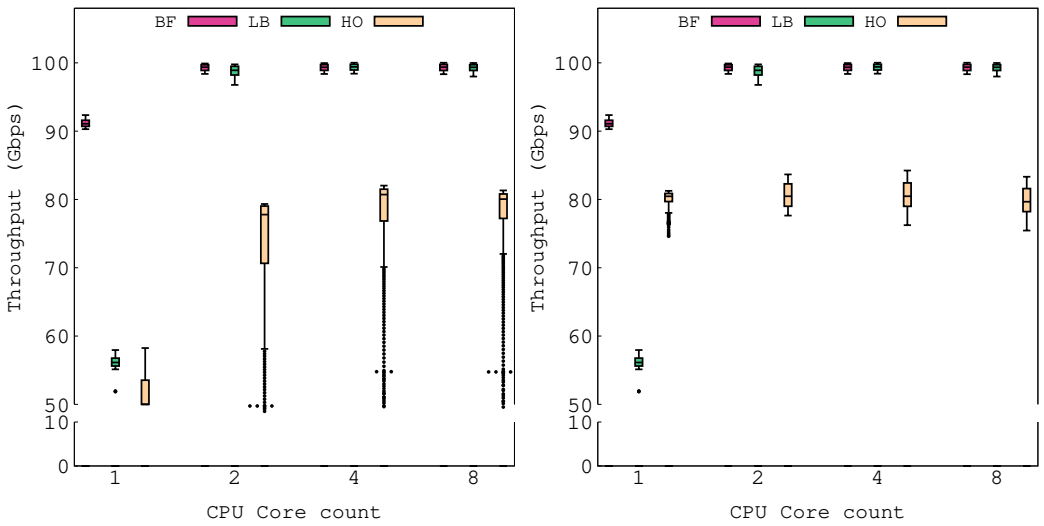
(d) 16M Hash table entries, 2M Flows, Run 2

Figure A.1: Throughput vs CPU core count



(a) 4M Hash table entries, 8M Flows, Run 1

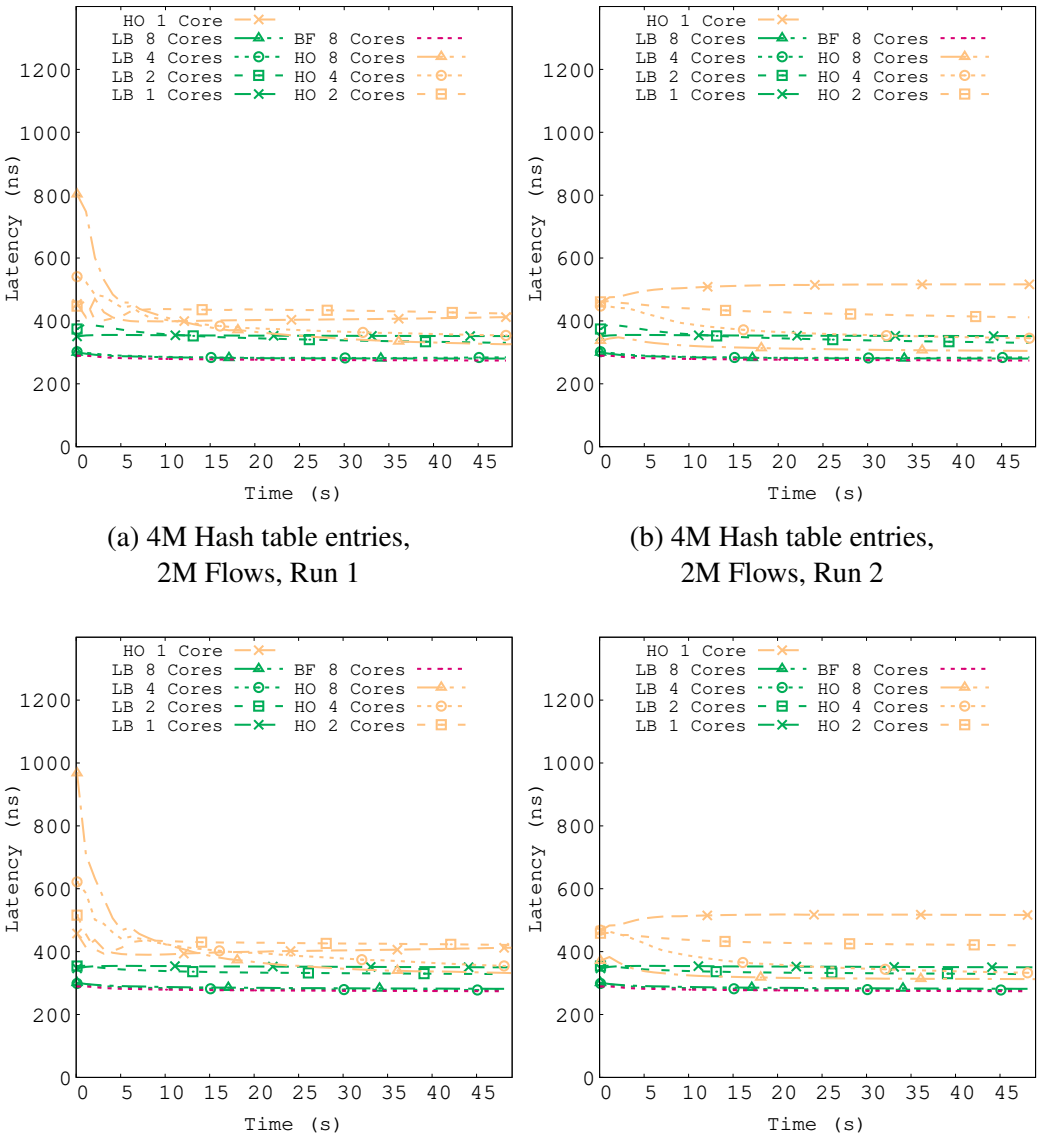
(b) 4M Hash table entries, 8M Flows, Run 2



(c) 16M Hash table entries, 8M Flows, Run 1

(d) 16M Hash table entries, 8M Flows, Run 2

Figure A.2: Throughput vs CPU core count



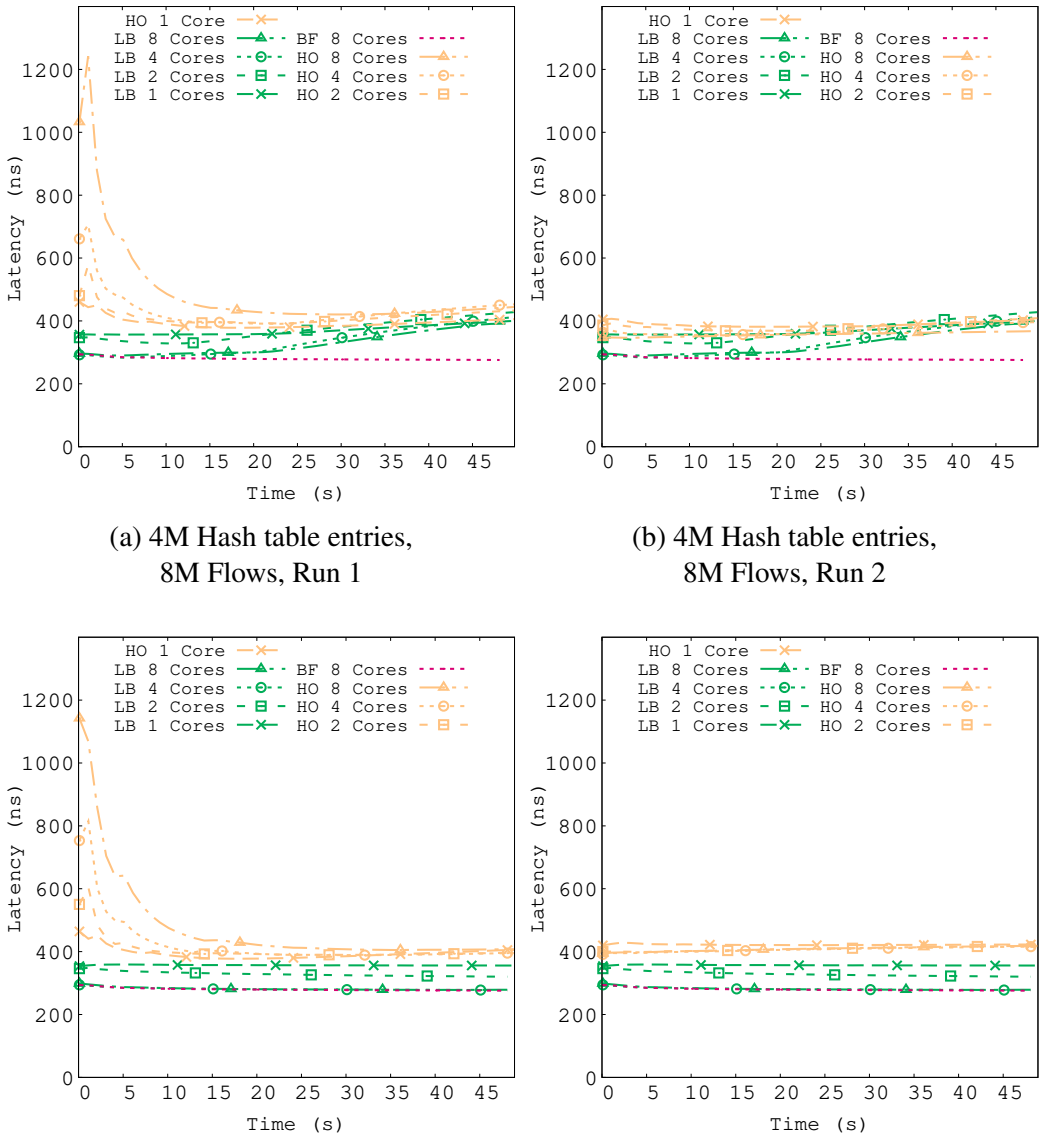
(a) 4M Hash table entries, 2M Flows, Run 1

(b) 4M Hash table entries, 2M Flows, Run 2

(c) 16M Hash table entries, 2M Flows, Run 1

(d) 16M Hash table entries, 2M Flows, Run 2

Figure A.3: Latency versus CPU core count.



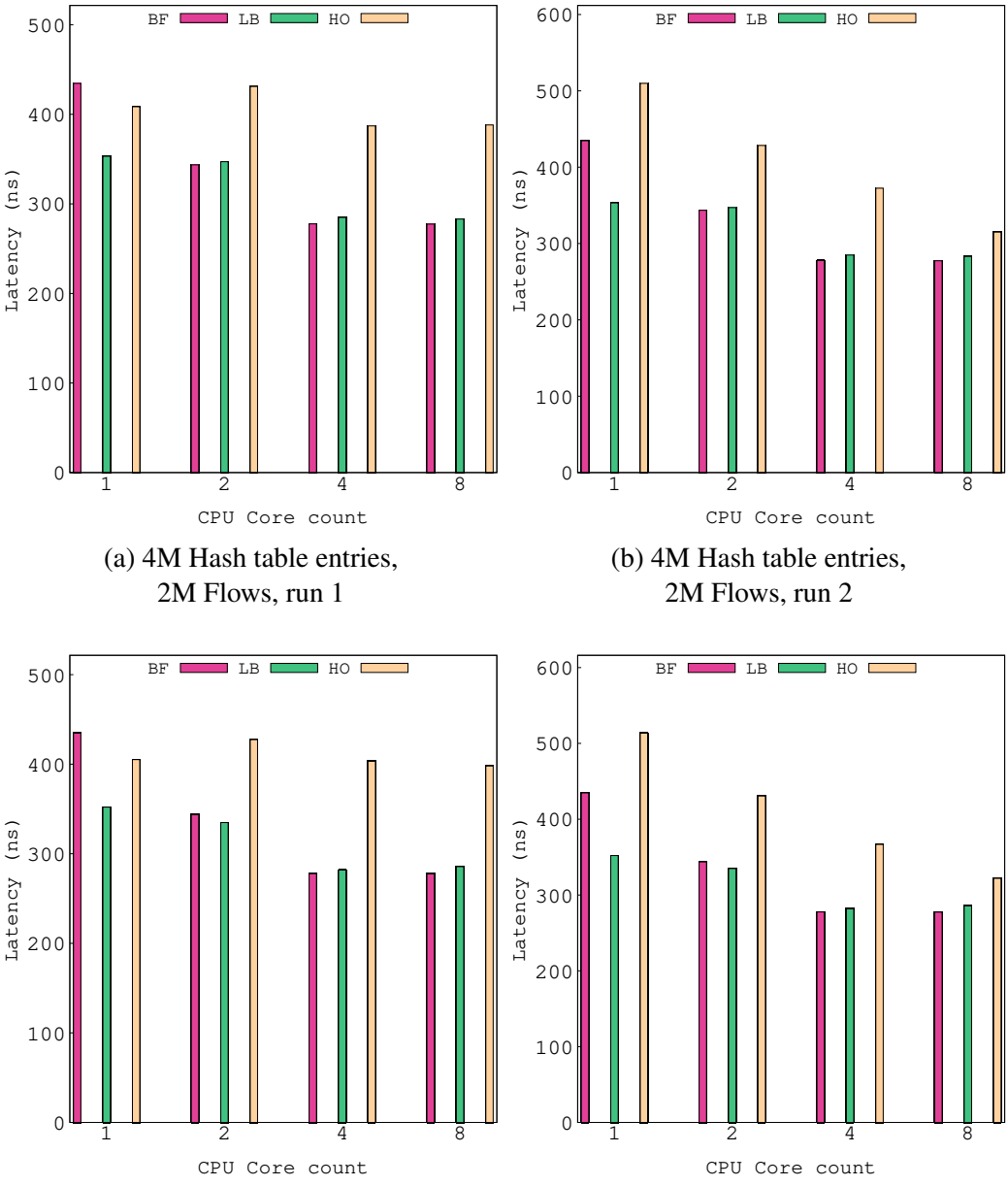
(a) 4M Hash table entries, 8M Flows, Run 1

(b) 4M Hash table entries, 8M Flows, Run 2

(c) 16M Hash table entries, 8M Flows, Run 1

(d) 16M Hash table entries, 8M Flows, Run 2

Figure A.4: Latency versus CPU core count.



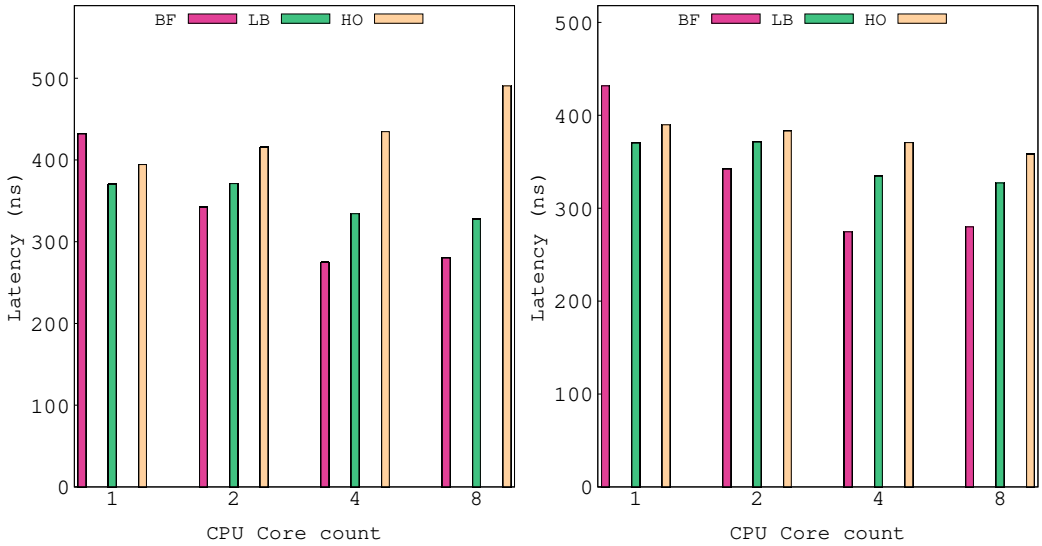
(a) 4M Hash table entries, 2M Flows, run 1

(b) 4M Hash table entries, 2M Flows, run 2

(c) 16M Hash table entries, 2M Flows, run 1

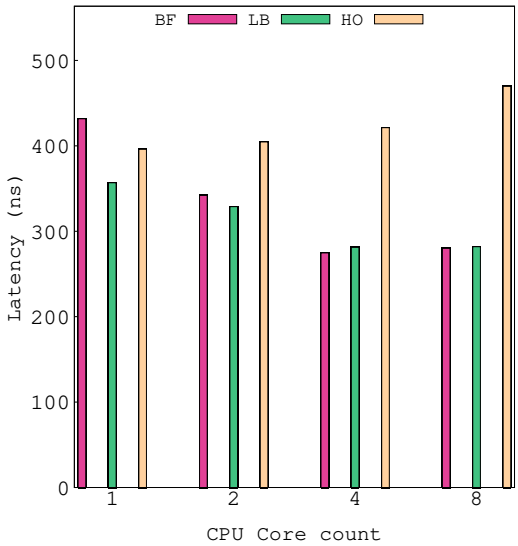
(d) 16M Hash table entries, 2M Flows, run 2

Figure A.5: Latency versus CPU core count.

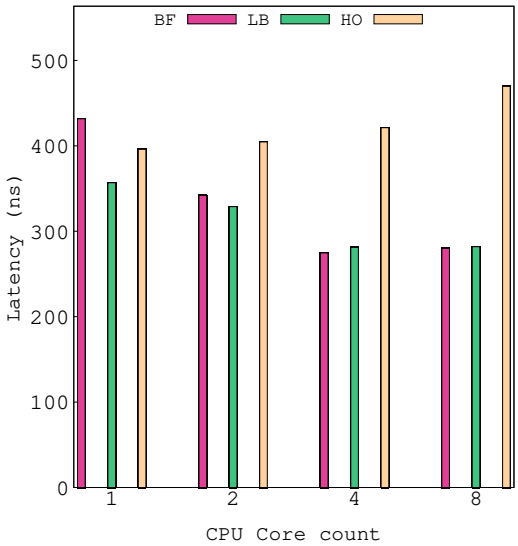


(a) 4M Hash table entries, 8M Flows, run 1

(b) 4M Hash table entries, 8M Flows, run 2

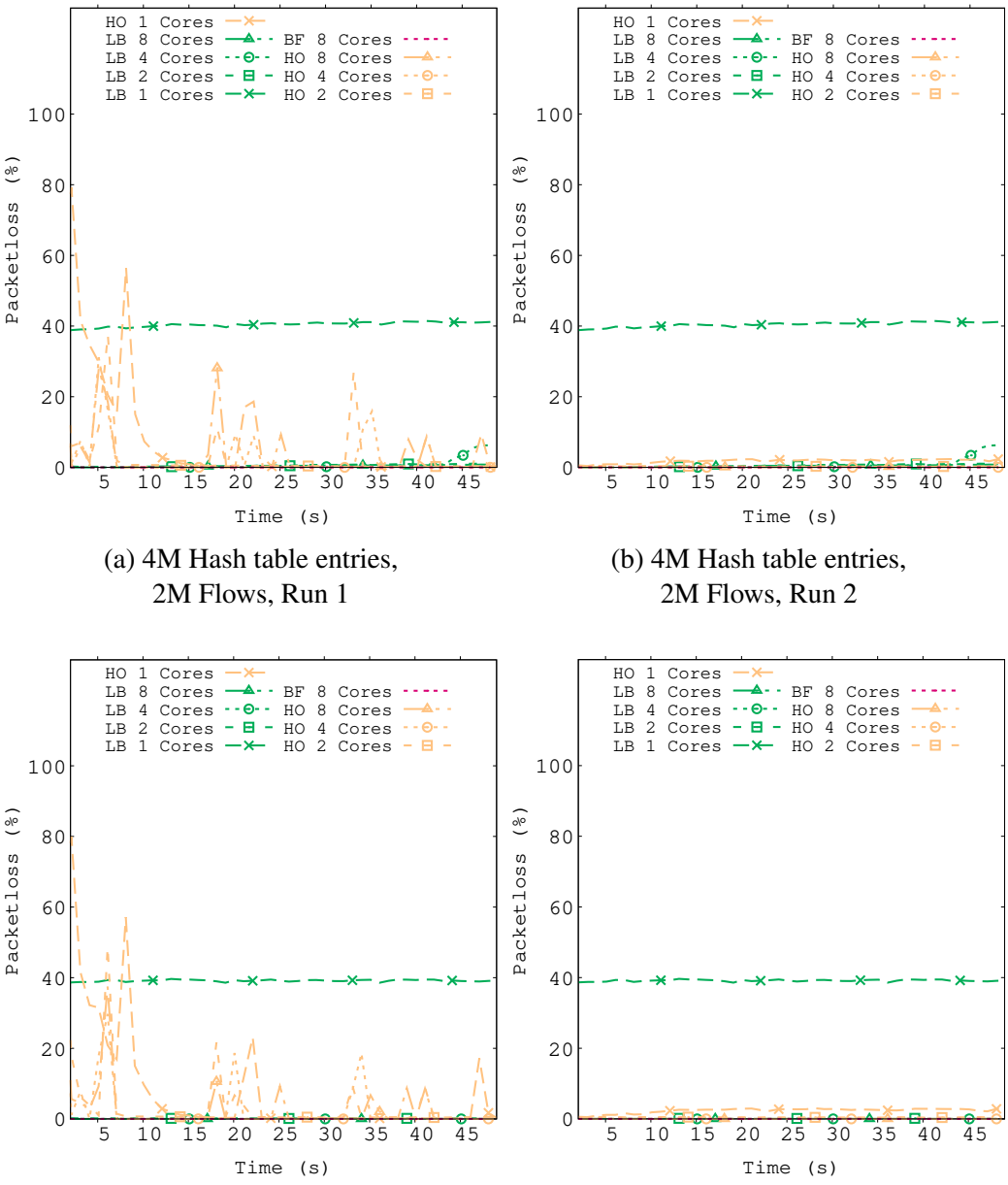


(c) 16M Hash table entries, 8M Flows, run 1



(d) 16M Hash table entries, 8M Flows, run 2

Figure A.6: Latency versus CPU core count.



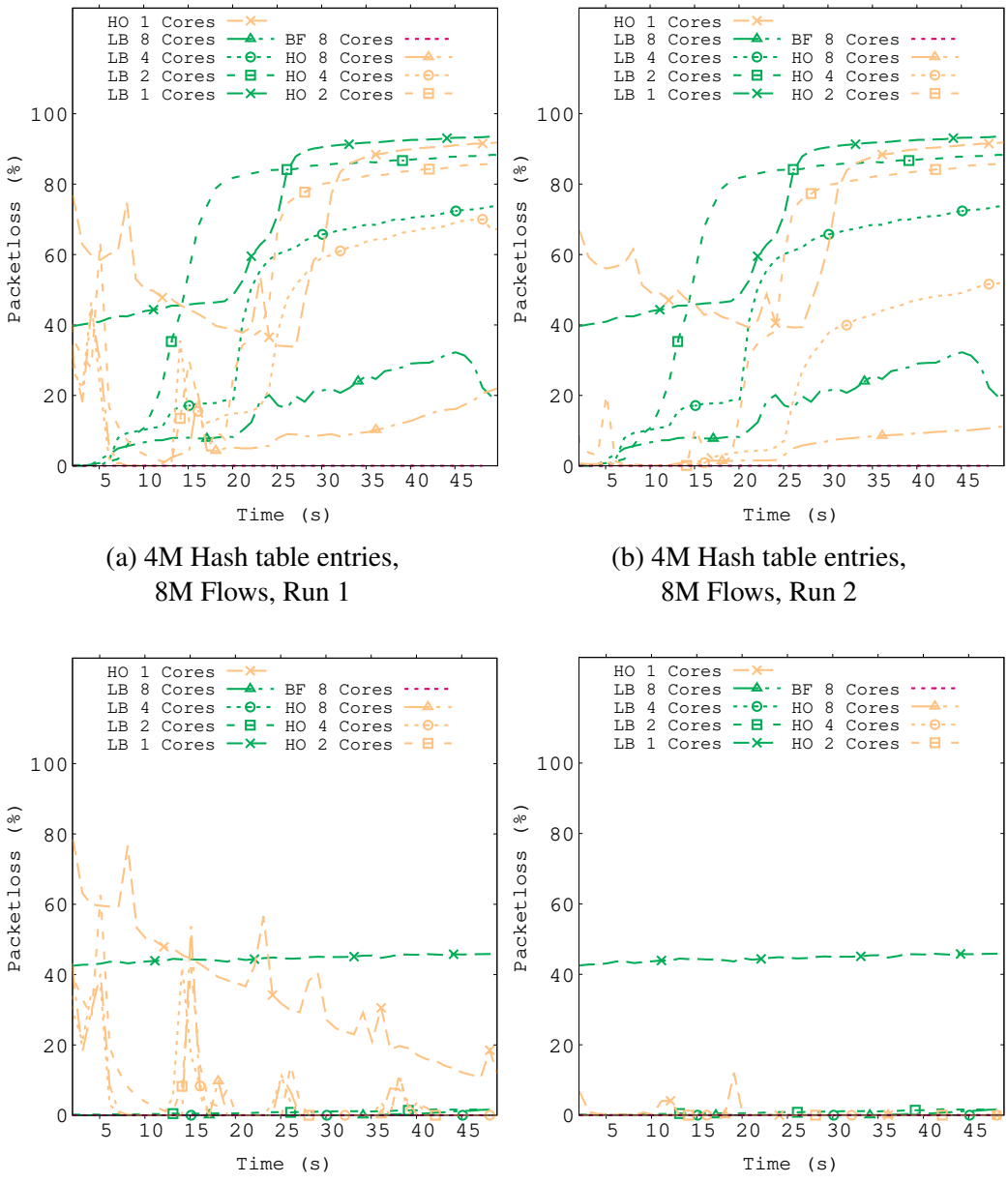
(a) 4M Hash table entries, 2M Flows, Run 1

(b) 4M Hash table entries, 2M Flows, Run 2

(c) 16M Hash table entries, 2M Flows, Run 1

(d) 16M Hash table entries, 2M Flows, Run 2

Figure A.7: Latency versus CPU core count.



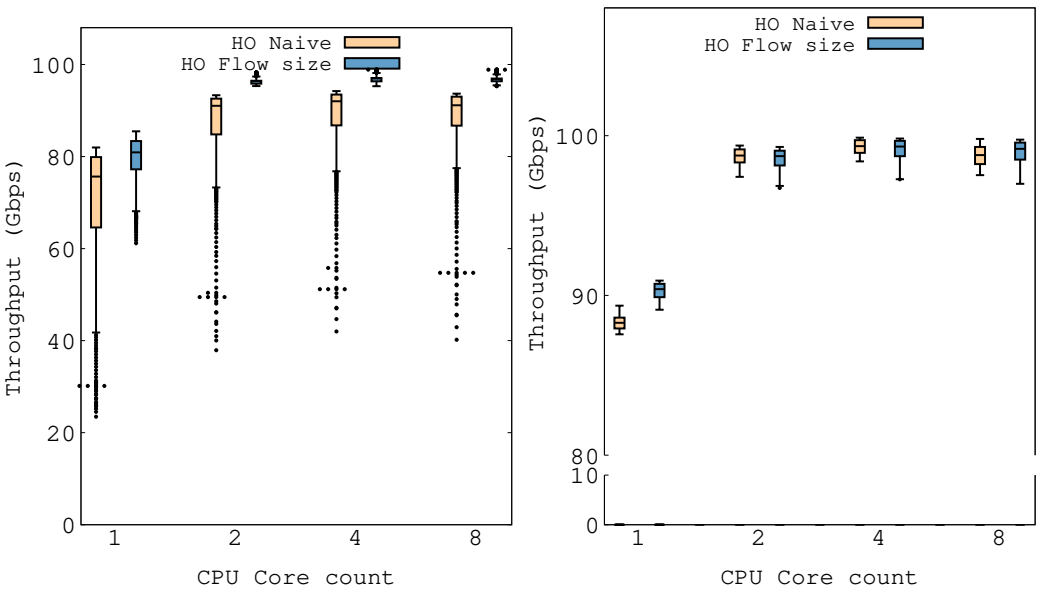
(a) 4M Hash table entries, 8M Flows, Run 1

(b) 4M Hash table entries, 8M Flows, Run 2

(c) 16M Hash table entries, 8M Flows, Run 1

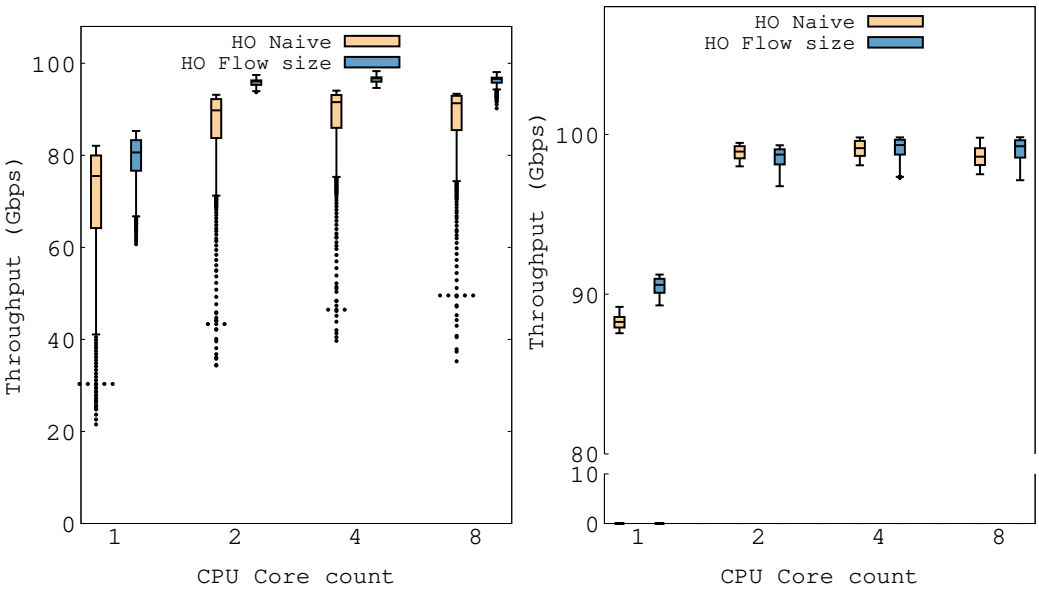
(d) 16M Hash table entries, 8M Flows, Run 2

Figure A.8: Latency versus CPU core count.



(a) 4M Hash table entries, 2M Flows, Run 1

(b) 4M Hash table entries, 2M Flows, Run 2



(c) 16M Hash table entries, 2M Flows, Run 1

(d) 16M Hash table entries, 2M Flows, Run 2

Figure A.9: Latency versus CPU core count.