Department of Informatics
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

BACHELOR'S THESIS IN INFORMATICS

Performance Analysis of VPP

Peter Okelmann

Technical University of Munich

Department of Informatics

Bachelor's Thesis in Informatics

Performance Analysis of VPP

Author:	Peter Okelmann
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisor:	Paul Emmerich
	Dominik Scholz
Date:	April 15, 2019

I confirm that this Bachelor's Thesis is my own work and I have documented all sources and material used.

Garching, April 15, 2019

Location, Date

Signature

Abstract

With high performance, low level network packet processing frameworks like DPDK, software routers can reach throughput speeds almost saturating 10GbE links with minimum sized packets using a single CPU core. Examples for this are open source projects like Vector Packet Processing (VPP), FastClick or MoonRoute. This paper analyzes how VPP can reach it's high packet throughput rates and describes possible bottlenecks. Tests to measure it's performance measurements are conducted and explained in detail. Results are presented for layer 2, Internet Protocol version 4 (IPv4), Internet Protocol version 6 (IPv6) and Virtual Extensible LAN (VXLAN) encapsulation scenarios. Especially different Forwarding Information Base (fib) sizes are tested to find out whether VPP is capable of routing the Internet. Forwarding latencies are analyzed and a brief comparison to other software routers will be made. Furthermore many measurement results are explained by creating models.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A world of microservices counting on virtualization is more and more interested in being able to move dedicated hardware into virtualized spaces. But especially high speed networking is traditionally bound to specialized hardware, because many software routers can't keep up performance-wise.

There are software routers though which focus on performance and are improved in many aspects. Especially frameworks like DPDK [13] help in the making of fast software routers which can run on affordable commodity hardware. One of the routers built on that framework is VPP which was developed by Cisco and is now open source and maintained by The Fast Data Project (FD.io). VPP can be virtualized and has packet input nodes for NIC's handled by The Data Plane Development Kit (DPDK) (dpdk-input), but can also use virtual links like virtual Linux interfaces (tuntap, af-packet-input, tapcli) or memory map based interfaces (memif, netmap, vhost-user) which allow for good interoperability with local virtualized systems. Therefore it is promising for highly virtualized environments.

There is little independent performance analysis of VPP though and none that compares it to other software routers. Therefore this work presents well documented [5] performance tests and results of VPP in different scenarios to enable comparability to other software routers. The tests procedures are as far as possible automated, to be as reproducible as possible. The resulting data is then analyzed to learn about the behavior of VPP and it's performance bottlenecks.

First, this paper will introduce related work about existing methods for router performance analysis and about existing benchmarking results for VPP in particular. Then challenges and performance critical aspects of software routing and VPP are analyzed.

In section 5 the methodology and the executed tests are described in detail. This paper presents measured performance for scaling on multiple cores, impact of varying CPU clocks, IP version, layer 2 fib and layer 3 fib sizes and tasks like VXLAN encapsulation. Finally the results are evaluated to create a model to describe VPP's performance behavior.

Chapter 2

Background

2.1 Core Features

Core features of VPP are...

... it's vectorized processing of packets in badges as the name indicates. This allows for better optimization.

... it's support for DPDK with it's fast, user-space NIC drivers. With it come also virtual packet interfaces for fast links to local virtualized systems.

... it's modular packet processing graph. Specific features are implemented in nodes for example like ip4-lookup, ip6-lookup or vxlan4-encap and vxlan6-encap. New features can be added with plugins extending the packet processing graph by new nodes, adding new processing paths and adding new CLI commands.

2.2 Releases

Each release (such as v18.10) contains the following pre-release milestones: With the first label "F0" the API is frozen and development shall aim for stability. For the second label "RC1" only bug fix changes are accepted and a first artifact is released. Then the iteration repeats and a second "RC2" version is released. Those pre-releases are marked with a tag containing the version number and the iteration appendix (v18.10-rc1). The final release will have no extra tags in their name. The git tag marks the official release, but the version-specific branch can still receive occasional updates. [19]

The oldest version of VPP available in the GitHub repository is from 2016. Unless otherwise specified this performance analysis used the v18.10 branch from 14th of December 2018[1].

2.3 Use Cases

A developer once formulated the design goal of VPP: It shall seamlessly integrate into existing Software Defined Network (SDN) stacks and it shall "become the foundation of the future cloud native services" [6]. That is, because VPP can serve in quite some different scenarios and can connect to different control plane tooling:

2.3.1 Data Plane

The internet the BGP table size grew to 512,000 prefixes in 2014 and it continues growing. As VPP aims to be used for routing tasks, it has to perform well with big routing tables. [1]

For handling high loads VPP helps spreading the computation intensive frame processing tasks to multiple CPU cores. Qosmos for example uses VPP to implement deep packet inspection [4].

Viosoft on the other hand uses VPP in the context of embedded systems which might require stable or short latencies [12]. Therefore latency behavior represents an important aspect of performance analysis, too.

2.3.2 Control Plane Integration

VPP integrates into open source control plane projects: OpenDaylight software can do some VPP management using an OpenFlow interface via Locator ID Separation Protocol (LISP) flow mapping [15]. Using control plane software like OpenDaylight, VPP also integrates into OpenStack Neutron, a networking management software for the OpenStack cloud computing platform [25].

Furthermore it can be assumed that Cisco's Vritual Provider Edge (vPE) product group uses VPP, because the term "vPE" can be found 39 times in the source code and many

[1] commit a8e3001e68d8f5ea6cf526b131c92f5993597f81

additional times in the official documentation [24]. vPE is a collection of Cisco's concepts for SDN, virtualized data centers and cloud native environments called Virtual Multiservice Data Center (VMDC) 1.x to 4.x. Those vPE concepts are used in several products [23]:

- Cisco VSG (Virtual Security Gateway)
- Cisco ASA 1000v, a Cloud Firewall
- Cisco vWAAS (Virtual Wide Area Applications Services)
- Citrix ADC VPX, a SDN suite

Chapter 3

Related Work

There has been a previous work analyzing and describing the design, architecture and performance of VPP [14]. It presents measurement results mainly regarding very specific properties like packet vector size or other optimization strategies but lacks results about the impact of hash table sizes. This paper on the other hand presents detailed test results for the layer 2 fib, IPv4 fib and IPv6 fib which allow for a quantitative performance comparison to other software router in a routing scenario.

For [14] the VPP nodes were extended to collect stats, such as perf does, for all processing functions individually which gives further insights. This paper on the other hand (see chapter 5) uses perf-record and perf- report, to compare used CPU time of VPP functions.

There is a paper [18] which aims to do RFC2544 compliant benchmarks with FreeBSD, Linux and an off-the-shelf MicroTik router. It summarizes existing methodology and benchmarking systems and presents relevant parts of the results. There is also a detailed analysis [17] of the Linux network stack as a router which presents a model for upper bound throughput predictions.

Another paper [9] presents benchmarking results about the MoonRoute software router and some comparative data to FastClick DPDK, Click DPDK and Linux 3.7.

There is also detailed analysis of the DPDK framework, comparison to other frameworks and a model describing it's performance. [10] Another work discusses more generally different ways of implementing fast user space packet processing and focuses especially on used concepts and optimizations. [2]

Unfortunately there is no universally fitting and established benchmarking routine. Even though RFC2544 [21] describes "Benchmarking Methodology for Network Interconnect Devices", many of the suggestions done in it are not relevant or applicable to software routers like VPP. [18]

For example it stresses procedures to test different Ethernet frame sizes. Frame size is typically no bottleneck for software routers and thus receives little attention in this paper. [7]

Furthermore the RFC states: "At the start of each trial a routing update MUST be sent to the DUT" [21]. Adding up to several million routing entries using networking protocols before every test run doesn't appear to be a feasible approach. Instead VPP's CLI is used to add all table entries using a single command.

Generally speaking, the RFC's requirements towards the duration of test runs exceeds what was done for this paper. According to the RFC a test run should take longer than a minute - latency measurements shall only measure timings of a single packet per minute. Providing a constant level of CPU performance for example by disabling Intel's turbo-boost, VPP stabilizes after only 10 to 20 seconds of receiving load which allows for way shorter test runs.

RFC 2544 uses some definitions from RFC1242 [20] like "Throughput": RFC 1242 defines throughput as the highest throughput without any packet loss, which is not the same as the maximum measurable throughput. This paper refers to throughput as the maximum measurable throughput if not specified otherwise.

For deeper insights into how the throughput and latency is measured by MoonGen, a tool used for this paper, [8] gives a lot of insight for example about the time measurement accuracy.

Chapter 4

Analysis

4.1 Experiment Setup

Each device in the testbed has at least three active Ethernet links: One connecting it to a management server, and two connected to the corresponding DUT or LoadGen. The management server (kaunas) uses Plain Orchestrating Service (pos) [11] to set boot images, boot, run setup and testing scripts on the testbed devices and collect script output and artifact files containing test results.

As figure 4.1 shows, each test contains two devices. The DUT, running VPP, and the LoadGen, running MoonGen, generating packets and sending them to the DUT to test VPP. VPP then forwards the packets back to the LoadGen which allows the LoadGen to create accurate throughput and latency measurements.

On the DUT some debug information like error counts are being gathered from the VPP CLI interface and Linux perf tools [16] attach to the VPP process to collect time spent per symbol and stats like CPU cycles or cache misses.

4.2 Possible Bottlenecks

4.2.1 Physical Link Speed

Running VPP with only one worker, the physical link speed is never the limiting factor. Usually this is 10GbE in this paper. Only with multiple worker scenarios the 10Gb/s mark is quickly reached. A simple way to address this is by testing with bidirectional

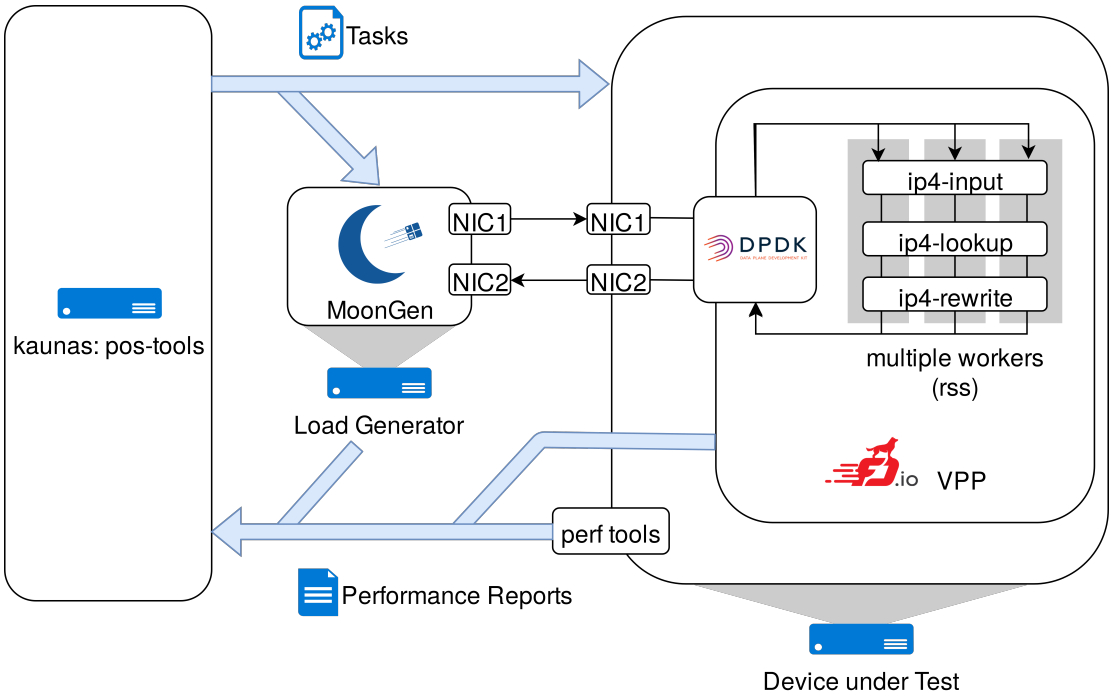

Figure 4.1: Experiment setup for testing VPP with "kaunas" management server, LoadGen and DUT

traffic - sending load to both of DUT's NICs and expecting VPP to forward it to both NIC's again. This way, the load on a single CPU core to be tested can be doubled, without actually requiring faster links. Other options are to reduce CPU clock speed or increase the CPU load artificially on a per-packet basis, in order to create another, even more limiting bottleneck on the CPU. The obvious solution, using a 40GbE link with more bandwidth, was used for this paper.

4.2.2 LoadGen Speed

Another limiting factor could be the load generator's speed of generating load. MoonGen, the LoadGen used for this paper, can saturate a 10GbE link with minimum sized Ethernet frames for all tests conducted. For a simple unidirectional 10GbE link this results in 14.88 Mpps [7].

Using faster 40GbE with Intel XL710 NICs showed a soft throughput limit of about 22.81 Mpps (min. frame size, unidirectional, standard deviation: 0.22). This was observed with an Intel E5-2620 v3 (2.40GHz). This limit varies when increasing the frame size. Previous work showed that the cause for this are bottlenecks inside the NIC. [8]

4.2.3 Internal Bandwidth

Inside the DUT there is first of all the PCI bandwidth. All tested NIC's are connected by a PCIe 2.0 x8 link with a maximum transfer rate of 32Gbit/s per direction. In theory a 40GbE link could exceed this transfer rate. Because the tests in this paper are only done with small packet sizes, the maximum reached packet generation rate of around 22.8 Mpps results in significantly less than 32Gbit/s. For bigger frame sizes, the width of PCI interfaces will have to be increased, because otherwise it will become a bottleneck for 40GbE links.

Secondly the Ethernet frames have to be moved to the main memory. It typically supports between 1333 MT/s (the 10GbE system) or 2133 MT/s (40GbE system) transmitting 64bit per message. This results in a transfer rate of 85 GBit/s and 136 GBit/s. This should be sufficient for 10GbE networking, 40GbE links require at least 2133MT/s memory.

4.2.4 CPU Time per Packet

As section 6.1 will show, the CPU time spent per packet is the biggest bottleneck. The following subsections describe how this issue is dealt with in VPP.

4.3 NICs with DPDK and VPP

For VPP to utilize NIC's up to their specified line rate, it relies on DPDK. This introduces a few noteworthy optimizations:

RSS (Receive Side Scaling): When the NIC receives a packet, it can be configured to hash over specified packet header fields to assign the packet to a receiving queue. The idea is, to efficiently separate for example different IP traffic flows early which can then be processed by several VPP workers concurrently and independently. [14]

Zero-Copy: Because memory bandwidth is actually limited, it is important not to move frames around unnecessarily. Therefore the NIC copies the received frames to memory shared with VPP which is their final fixed location. [14]

Busy Polling: Newer VPP versions have options to be either notified about new packets in the shared memory via interrupts, or alternatively and better suited for high

performance applications, VPP workers can be busy polling. The latter option reduces performance dependencies on the hardware and kernel. [28]

4.4 Packet Processing Graph

VPP's packet processing is defined by the packet processing graph. Figure 4.2 illustrates selected packet paths important for this paper. Depending on the plugins and configuration in place, this graph can be customized for example to replace specific software functions with hardware accelerators. Depending on the initial packet parsing which happens in the "dpdk-input" and "l2-input" nodes, the packets traverse the graph over their respective paths.

xconnect: Static layer 2 connections between physical links (xconnect) are parsed and passed through the dpdk-, ethernet- and l2-input nodes and only use the l2-output node to be sent to the target device.

Bridging: With bridged configurations, packets are treated the same as xconnect traffic, but there is an additional l2-fwd node which looks up the destination in the layer 2 fib. Packets for this node are processed by the `l2fwd_node_inline` function implemented in `vpp/src/vnet/l2/l2_fwd.c`.

IPv4: The IPv4 path shares it's packet header parsing with the layer 2 processing graph. After the parsing of a User Datagram Protocol (UDP) packet, the ip4-lookup and ip4-rewrite nodes implemented by the `ip4_lookup_inline` and the `ip4_rewrite_inline` function in `vpp/src/vnet/ip/ip4_forward.c` take care of looking up the next-hop destination and rewriting the packet header. Their predecessing node (ip4-input) is also predecessor for other ip4 protocol nodes like for example ICMPv4. A lot of the header parsing happens before the ip4-input node though, even if this information may not be needed for the follow-up nodes. This means for example, that VPP even in layer 2 bridging configuration is slower with switching UDP packets compared to switching packets of an unknown ethertype.

IPv6: The ip6 packet processing path differs completely from the previously discussed paths, because it implements it's own parsing in the ip6-input node. For lookup and forwarding purpose, there follow an ip6-lookup and an ip6-rewrite node, similar to the

IPv4 path. Those nodes are implemented by the `ip6_lookup` and the `ip6_rewrite` function in `vpp/src/vnet/ip/ip6_forward.c`.

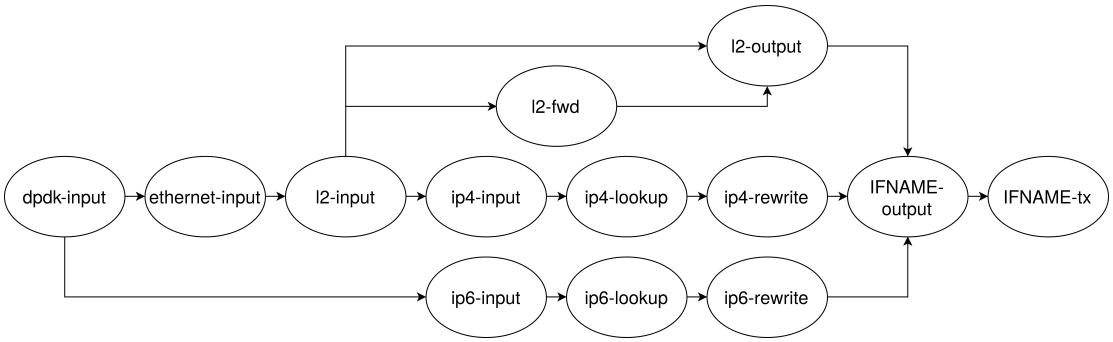

Figure 4.2: VPP packet processing graph for xconnect, bridging, IPv4 routing and IPv6 routing. Other paths are left out.

4.5 Forwarding Data Structures

VPP implements several data structures for it's packet forwarding. A BiHash table is used for the layer 2 fib and the ip6 fib (`vpp/src/vppinfra/bihash*`) [3]. IPv4 fib lookups use a mtrie (`vpp/src/vnet/ip/ip4_mtrie.h`).

l2-fib: For packets reaching the l2-fwd node, the destination lookup is done for example by `l2fib_lookup_4` in `vpp/src/vnet/l2/l2_fib.h`.

Each lookup takes a 64 bit `l2fib_entry_key_t` containing the mac address and the bridge domain index of the incoming interface. It returns a 64 bit `l2fib_entry_result_t` with the relevant destination information.

IPv4-fib: The IPv4 lookup is implemented by `ip4_fib_table_lookup` in `vpp/src/vnet/fib/ip4_fib.h`. The fib struct `ip4_fib_t` contains two data structures. A list of hash tables (implemented in `vpp/vppinfra/hash.h`), one for each IP prefix length, which is only used for Command Line Interface (CLI) status creation and to enforce unique inserts. The second data structure is used by the ip4-lookup node for IPv4 fib lookups. The mtrie lookup results in an adjacency index which is then used by the ip4-rewrite node to rewrite the destina-

tion. Therefore the rewrite node gets the `ip_adjacency_t`[1] information from the respective adjacency vector according to the adjacency index.

IPv6-fib: The IPv6 fib lookup itself is done by the ip6-lookup node and implemented by `ip6_fib_table_fwding_lookup` in `vpp/src/vnet/fib/ip6_fib.h` and uses the BiHash table for lookups in the ip6-lookup node. The lookup returns an 32 bit adjacency index which will be used by the ip6-rewrite node to get the next packet destination from the `ip_adjencency_t` vector, just like the ip4-rewrite node does.

4.6 Vectorization

Vectorization refers to VPP collecting incoming packets in batches. The input batch size is defined at compile time by `VLIB_FRAME_SIZE` and per default set to 256. This is only the maximum size though. When packets don't arrive fast enough, batches are closed and submitted to the processing graph, before the size limit is reached. Nodes which classify packets might split input batches to allow packets to traverse to different subsequent nodes. In this case the batch size shrinks further.

This procedure reduces the number of *instruction cache misses*, because a new node and it's associated function is loaded into the cache only once for every batch, instead of once per processed packet, as long as the node is small enough to fit in the cache. The node switching overhead is therefore shared among all packets in a batch.

Additionally input batching allows for better *data prefetching*. When each packet traverses the whole graph in a single run, nodes never know what data the next node will need to be prefetched. With input batching a single node processes many similar packets in a row and can thus efficiently prefetch the data it needs for the next packets of the current batch. Only the first few packets won't be prefetched.

Comparing input batching to *FastClick batching* there are two major differences: "Nodes in FastClick implicitly process packets individually, and only specific nodes have been augmented to also accept batched input" [14]. Additionally VPP has better optimized buffer management and allocation for input batching for example through reuse of already allocated buffers. [14]

[1] `ip_adjecency_t` is defined in `vpp/src/vnet/adj/adj.h`

Furthermore the packet batch entering a node can be processed in batches, too. The packet processing function of a graph node may take typically up to four packets as parameters to process them simultaneously. This results in the minimum `VLIB_FRAME_SIZE` being four. Not all nodes implement this feature, sometimes called "*multi-loop*", though because not all processing functions profit from it: While the profit in the xconnect processing path is negligible, the IPv4 lookup node is 23% faster with quad loop than with single loop processing. The IPv6 lookup node is only 5% faster. [14]

Chapter 5

Methodology

In this paper the performance of VPP is quantified by throughput and latency which can be measured by the MoonGen [8] load generator. To help identifying bottlenecks and limiting factors, perf is used to measure metrics like cache misses and clock cycle counts. Furthermore VPP's error counters are collected after each run, to detect problems in the packet processing tests.

5.1 Measurement Results

5.1.1 MoonGen (Throughput and Latency)

The DUT sends test traffic using MoonGen. There is a lua script determining MoonGen's behavior for generating layer 2 Ethernet packets, another one for layer 3 IPv4 packets, one for IPv6 packets and one for VXLAN packets. All but the latter can be given command line parameters to cap the packet rate, change the packet size, the destinations, the sources and the amount of destinations to send to.

Additionally the scripts grant VPP some warmup time: Only when the first packet is successfully forwarded, the actual test begins. This is necessary, because VPP v18.10 tends to drop packets in the interface-tx node for a few seconds when it is not warmed up which could drastically distort the average throughput and latency. After the other components have settled into the high load situation which takes around a second, the throughput average stays about constant and can thus be properly described by the average and standard deviation.

To measure the latency, occasional, timestamped packets are sent by MoonGen between the load generating packets. This is used to calculate average and standard deviation and create a histogram of latencies.

5.1.2 perf

perf-stat: The command `perf stat` is used to attach to the `vpp_main` process. This allows collection of hardware counter information about different types of cache events, cpu cycles used etc. The stats of all children are added up to the main thread's ones.

perf-record/perf-report: The command `perf record` is used to attach to the first worker thread of VPP or if VPP runs single threaded, to the main thread. This records the CPU time spent per function (symbol). It is important to connect to a thread processing the packets in VPP's graph (not necessarily the main process) to get significant results for identifying performance limiting factors. After the test run, the record has to be parsed with `perf report` to associate function names to the symbols.

Performance Impact: Those two tools run during the test and can therefore impact the DUT's performance. For IPv4 forwarding the impact can be up to 8% decrease from the original packet throughput rate. That's why the perf tools are only active for graphs showing cache misses or time spent per function.

5.1.3 VPP Counters: socat

Additionally VPP internal counters, usually used for error counting, are saved into the test's report for test quality assurance and debugging. The counters can be read via the `show errors` CLI command. Therefore VPP is configured to listen at a Unix file FIFO for it's CLI. This interface can be conveniently used with the tool "vppctl" in VPP v18.10 to connect and interact with the FIFO. In earlier versions like v16.09 this tool worked differently. In order to be able to read those counters nevertheless, the tool "socat" is used to connect to the CLI FIFO file.

5.2 Test Procedure

One test run tests one VPP configuration. After each test run, VPP is being restarted. After each restart, the performance of VPP varies significantly more than within one

test run. Therefore in order to get an insight in how spread the variance is, each test run is repeated 6 times. Only histogram latency measurements rely on a single run.

As figure 5.1 shows, the DUT and LoadGen first read the remote configuration file, after the management server has started the test. Afterwards both devices configure the Intel pstate driver as described in section 5.5 and the DUT loads the kernel module for the NIC's. Then DUT and LoadGen are synchronized as the pre-test *setup is complete*.

Before starting the new VPP daemon, the old one is killed and some files which may contain global state are removed. The VPP daemon will take at most a few seconds to set itself up, so both devices are synchronized again and MoonGen is already being started, because it's setup takes longer. When MoonGen is set up and starts it's warmup LoadGen-phase, VPP is already online. *20 seconds* after MoonGen started, both systems are *running and have stabilized*.

Now, after both systems synchronized again, *perf data is collected* as described in section 5.1.2 for a timespan of 10 seconds. Then both systems synchronize again, VPP's counters are queried (section 5.1.3), MoonGen is killed and after giving it 10 seconds time to finalize the throughput and latency histogram, all files containing test results are uploaded to the management server.

Approximately 43 seconds after the test started, the final synchronization takes place an the *next run can be started*.

5.3 Configuring VPP

Configuring VPP consists of two steps: Setting the startup parameters which is done using a Configuration File (config) - and executing CLI commands to configure the routing settings on runtime. For the latter a file containing the CLI commands can be specified in the startup config to be executed.

The VPP starting scripts used for this paper therefore create temporary config and Execution file (exec) for example according to the number of IPv4 routes specified in the starting script's parameters. The IPv4 and IPv6 scripts also support arbitrary multicore (-pinning) configurations.

Adding millions of IPv4 or IPv6 entries to the layer 3 fib is no problem, because the CLI already has a command parameter to add any number of entries through a single command. The CLI commands for the layer 2 fib don't support this though. Adding millions of entries via the command line would be too time consuming. That's why

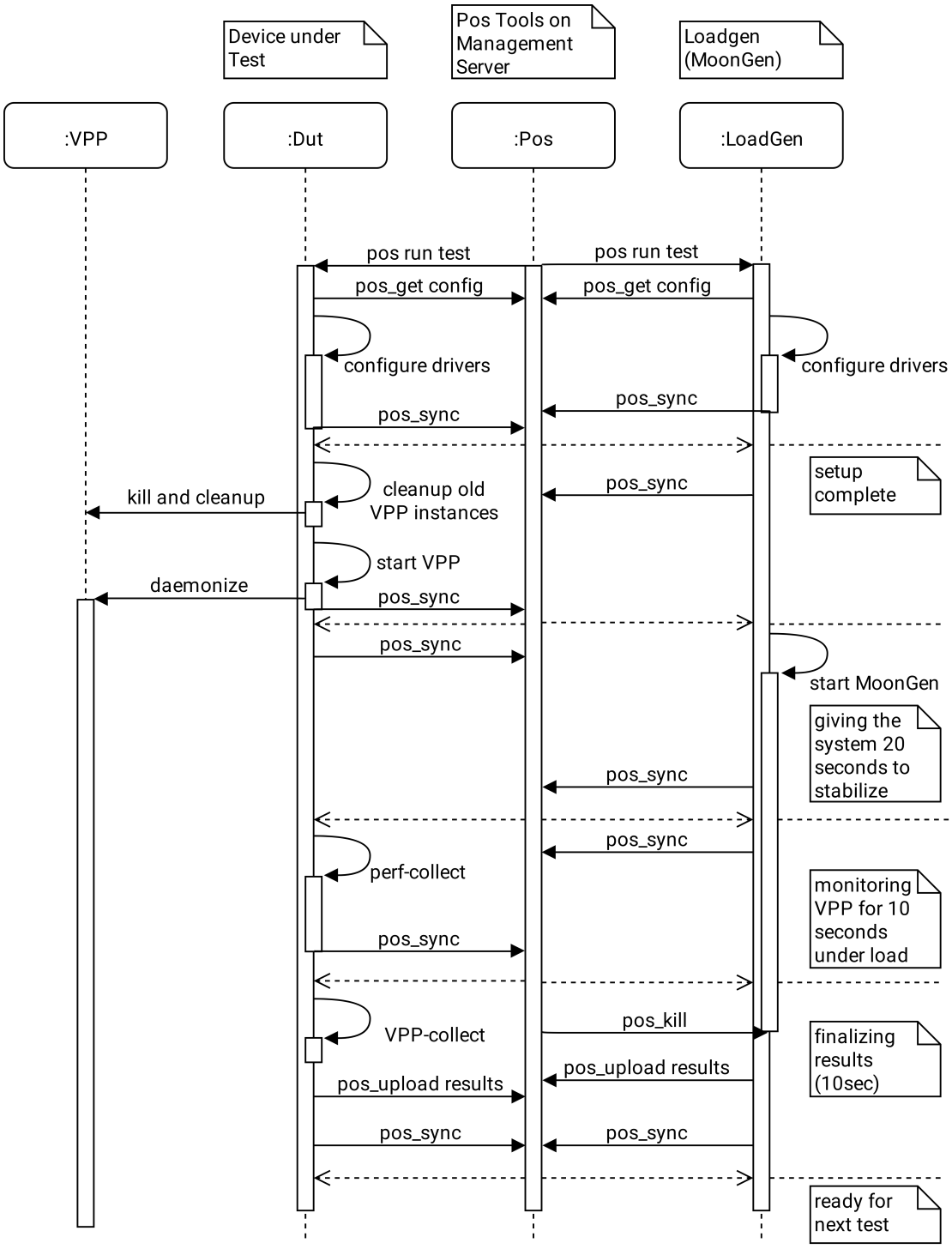

Figure 5.1: Sequence of a single test run.

this paper uses an own VPP plugin which extends the CLI by a command with this functionality. In terms of speed it is comparable with the built-in commands.

5.4 Tested Scenarios

For this paper there are many VPP configurations to test several scenarios:

XConnect: The most simple configuration is linking two network interfaces via cross connect (xconnect). This is a static mapping from one interface's input to another interface's output and vice versa.

L2 bridging: For layer 2 switching VPP can create bridge domains. The optional bridge features mac-learning and mac-aging impact performance when activated, even when not specifically tested. Other features like flooding settings have no significant performance impact when not directly tested. Therefore in section 6.1 results are presented with and without the impacting features enabled.

L3 routing: The IPv4 and IPv6 routing tests use multiple routes which all point to the same destination, conforming RFC2544.

VXLAN: Setting VPP up to encapsulate VXLAN packets takes among others the following steps:

1. Add a dedicated IP fib table and configure it to route to the VXLAN decapsulation endpoint.
2. Create a bridge domain belonging to the dedicated IP table.
3. Setup the tunnel using the bridge domain and the dedicated IP table.
4. Add the new virtual `vxlan-tunnel*` interface and all other necessary interfaces to the bridge domain and take care of all of their l2fib entries.

This means all packets arrive in the bridge domain, are forwarded to the virtual vxlan-tunnel interface, then encapsulated and sent to the vxlan-tunnel endpoint via IP.

5.5 Testbed Hardware Setup

Test are conducted on different test systems listed in table 5.1. All of the used systems (the LoadGen ones, too) are configured via the Intel Pstate driver to disable turbo boost in order to maintain a close to constant clock speed.

DUT	NIC Model	NIC type	RAM
klaipeda	Intel 82599	2x10GbE	DDR3 1333 MHz CL9-9-9-24
omastar	Intel XL710	2x40GbE	DDR4 2133 MHz CL15-15-15-35

DUT	CPU @ clock (GHz)	physical cores	L1 cache	L2 cache	L3 cache
klaipeda	Xeon E3-1230 @ 1.6-3.2	4	32k	256k	8M
omastar	Xeon E5-2630 v4 @ 1.2-2.2	10	32k	256k	25M

Table 5.1: DUT hardware: CPU model with minimum and maximum clock speed reachable with the Intel Pstate driver without turbo boost, NIC model and RAM timings

Chapter 6

Evaluation and Model

6.1 CPU as Bottleneck

Table 6.1 presents maximum throughput of all measured VPP configurations with minimum sized packets using a single VPP worker.

The results show, that the computationally least expensive scenario (xconnect) allows for the highest packet rates. Furthermore we see, that when reducing the CPU clock speed by 3%, the number of processed packets shrinks by around 3%, too. This indicates, that the CPU cycles are bottlenecking the packet throughput of VPP.

This means the packet throughput $f(c)$ can be approximated depending on the clock speed c for maximum packet rates p_{max} and the maximum clock speed c_{max}:

$$f(c) = (0.9 * \frac{c}{c_{max}} + 0.1) * p_{max}$$

This models approximates for 50% of the maximum clock speed a throughput of 55% of the maximum one. This does not exactly resemble the expected function $f(c) = \frac{c}{c_{max}} * p_{max}$ (half the performance at half the clock speed) which were to be expected if the CPU cycles were the only limiting factor. This expected behavior can be observed though with MoonRoute [9] which indicates another limiting factor, besides CPU frequency.

Scenario	1.6GHz (50%)	3.1GHz (97%)	3.2GHz (100%)
xconnect	7.34 (56%)	12.90 (98%)	13.20 (100%)
l2 bridge no features	6.69 (55%)	11.84 (97%)	12.18 (100%)
IPv4 1 route	6.03 (53%)	10.98 (97%)	11.28 (100%)
l2 bridge mac-learn, mac-age	6.05 (55%)	10.84 (98%)	11.09 (100%)
IPv6 1 route	5.38 (53%)	9.87 (97%)	10.14 (100%)
VXLAN encap	4.48 (55%)	8.15 (99%)	8.21 (100%)
IPv4 255k routes	4.19 (58%)	7.06 (97%)	7.25 (100%)
IPv6 255k routes	2.34 (62%)	3.72 (98%)	3.80 (100%)

Table 6.1: Maximum throughput (Mpps) in different scenarios with different CPU frequencies measured on the Xeon E3-1230 system.

6.2 CPU Scaling Model

Since the CPU is a bottleneck in all tested configurations, the ability to distribute the load over multiple CPU cores is essential. Not all VPP configurations support this though.

VPP has two concepts for using multiple cores: It has workers for simultaneous receiving and processing of packets over the processing graph - and it can be configured to use another set of workers for prioritized sending of packets (Hirarchical Quality of Service (HQoS) [26] [27]). Only the first type of worker can be used to enhance the performance of raw processing throughput though.

Many layer 3 processing nodes such as the IPv4 and the IPv6 processing path do support parallel packet processing using Recieve Side Scaling (RSS) (see section 4.3). For the scaling to have effect, the packets need different packet headers though, so that RSS header field hashing can assign them to different queues. Using four different source addresses per queue is sufficient to hit all of them well.

Layer 2 bridges on the other hand are completely single threaded. Because VXLAN needs a bridge to receive or send layer 2 traffic, it can't utilize multiple queues, too.

Scaling measurements are conducted with a 10GbE and a 40GbE link with high and low CPU clock speed. Tests begin with VPP in single threaded mode, having main core and worker in the same thread and continue with using unused physical cores for workers.

As figure 6.1 shows, switching from single threaded mode to a dedicated worker increases throughput slightly, but barely noticeable. Two cores are already enough to saturate

the 10GbE link. The virtual ("hyper threading") cores are used for workers three to six within the 10GbE system which results in lower performance gain and even slight performance loss in the end.

Using the 40GbE system at maximum CPU clock two cores can just not saturate the limits of the NIC. Both 40GbE scaling tests show that for a worker's maximum throughput w_t the overall maximum throughput depending on the number of workers w is $f(w) = w * w_t$. This holds true until line rate is hit at around two and four workers, respectively (using physical cores).

At around 20 Mpps the measured throughput can be unstable by around 0.4 Mpps, because of unstable MoonGen packet generation rates. This is due to the NIC performance limit described in section 4.2.2.

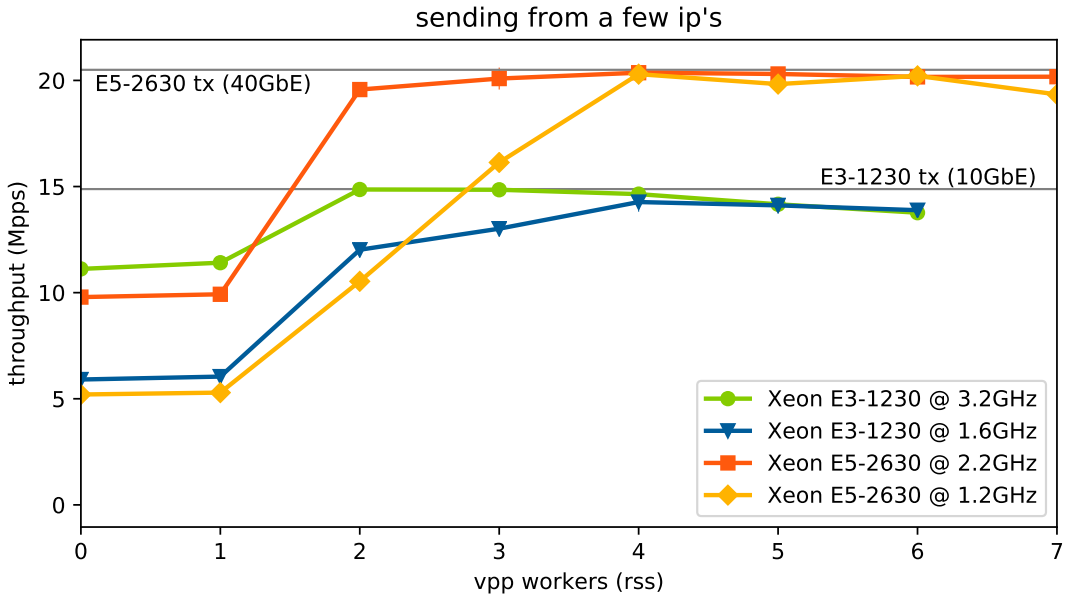

Figure 6.1: VPP CPU scaling with IPv4 traffic with 10GbE, 40GbE and different CPU clock speeds. All workers (besides E3-1230 core 3-6) use physical CPU cores. The 40GbE NIC bottlenecks at around 20Mpps (see section 4.2.2).

6.3 Lookup Performance Model

6.3.1 L2 fib

The layer 2 fib is tested by configuring VPP with varying amounts of l2fib entries and sending packets to random destinations using a single worker.

Figure 6.2 shows that with only one l2fib entry you get 0.5 Mpps more throughput compared with two l2fib entries. This is due to the simple caching mechanism of the l2fib-lookup function: It stores the last looked up information locally, so that when the next packet goes to the same destination, the local values can be used and a hash table lookup can be skipped.

From there on there are three key points when the CPU's L1, L2 and L3 cache are full. We can model the BiHash l2fib lookup table. It takes 64 bit values as input and returns a 64 bit result. Assuming the table only contains keys and values, the maximum fib size fitting into cache of n bytes is:

$$f_{l2fib}(n) = n * \frac{8}{2 * 64} = n * \frac{1}{16}$$

This formula is used to create the l1, l2 and l3 cache marks in figure 6.2 as a upper bound for how many fib entries could fit into the respective caches.

As figure 6.2 shows, the throughput drops before the L1 cache as the L1 cache misses start rising. With the Layer 2 cache mark the throughput drops further and more CPU time is spent inside the `l2fwd_node` function which does the lookup. Finally before the layer 3 mark, the respective data cache misses rise, the throughput drops further and close to 40% of the CPU time is spent inside the lookup node.

6.3.2 IPv6 fib

Figure 6.3 shows the results of IPv6 routing with different layer 3 fib sizes. It's maximum size is remarkably smaller compared to the layer 2 fib. Each lookup table result value is only 32 bit in size. Assuming the table only contains those values and keys the size of an IPv6 address, the maximum fib size fitting into cache of n bytes is:

$$f_{ip6fib}(n) = n * \frac{8}{128 + 32} = n * \frac{1}{24}$$

This means for a layer 3 cache size of 8MB (2^{23}B) at most 350,000 could fit into it. At this fib size we are off the charts though and there is no correlating change in throughput. Therefore the model for layer 2 fibs seem not to apply to IPv6 fibs.

Nevertheless it can be assumed the massive drop in performance from 20,000 fib entries upwards is because of the BiHash table lookups taking longer because of cache size limitations, since the big drop of throughput closely correlates to the layer 3 cache misses and the time spent in the lookup function.

6.3.3 IPv4 fib

While the layer 2 fib can contain over 2^{23} entries, tests show that VPP v18.10 stops working with more than 287,743 IPv4 fib entries. To be able to compare VPP to other software routers nevertheless, tests conducted with v16.09 show that this version is able to handle up to around 12,580,000 fib entries which is well above 2^{23}.

Figure 6.4 and 6.5 show test results of VPP v18.10 with up to 255,000 entries and of v16.09 with up to 10,300,000 entries. Both show similar behavior to the IPv6 fib tests with one big throughput drop towards the end of the graph.

6.3.4 Lookup Nodes as Bottleneck

With big lookup tables, the throughput drops a lot with every kind of lookup table. For every additional CPU cycle spent for the lookup in relation to the available cycles, the throughput will drop by the same ratio. Thus the maximum throughput f can be modeled depending on the relative time d_{lookup} spent in the lookup node function for a known maximum throughput t_{max} with one or two routes:

$$f(x) = t_{max} * (1 - d_{lookup})$$

The measurements closely resemble this model. They proof that the lookup is basically exclusively responsible for the performance decrease of bigger fibs and thus a limiting factor for scenarios with big routing tables.

6.4 Latencies

6.4.1 Average Latencies

Measuring the latency should be done with reduced offered load, because when offering the load at full link speed, the average latency hits a worst case which is orders of magnitude higher than at lower packet rates. Figure 6.6 shows the average latency and the packet loss at different packet rates relative to the No Drop Rate (NDR) for IPv4 routing with 1 and with 255,000 routes. It shows that the latency is around $5\mu s$ for low packet rates and slowly increases towards reaching the maximum throughput. When this maximum is reached, the packet drop rates immediately go up, because not all packets can be forwarded. This is also exactly the point where the latency explodes to around $600\mu s$ because packet queues fill up.

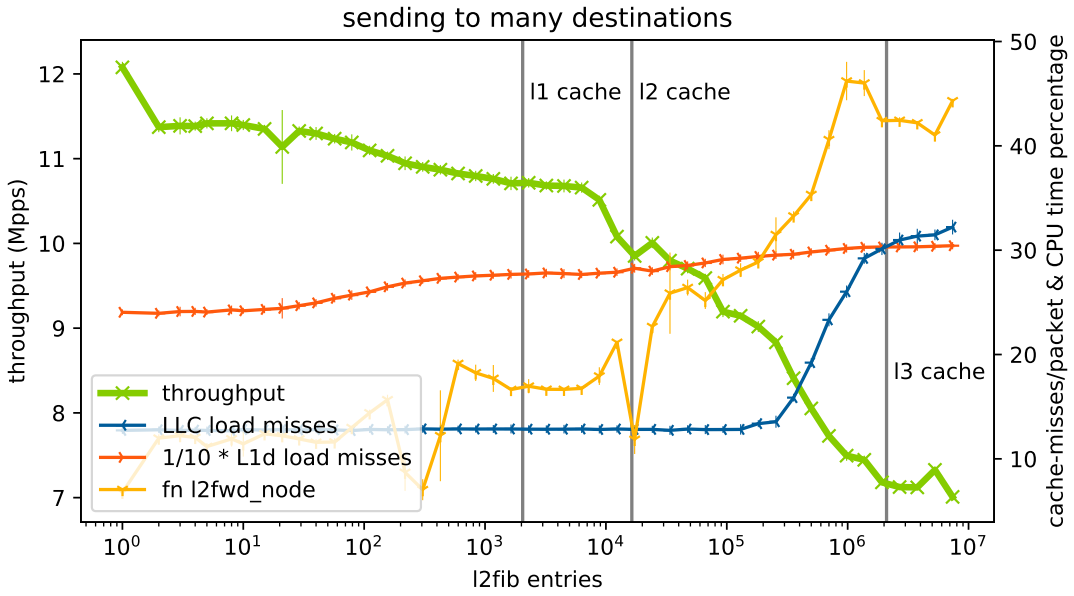

Figure 6.2: Testing VPP v18.10 with different layer 2 fib sizes. Throughput, Layer 1 data cache load misses (divided by 10) and Last Level Cache load misses per packet and the percentage of CPU time spent in selected functions.

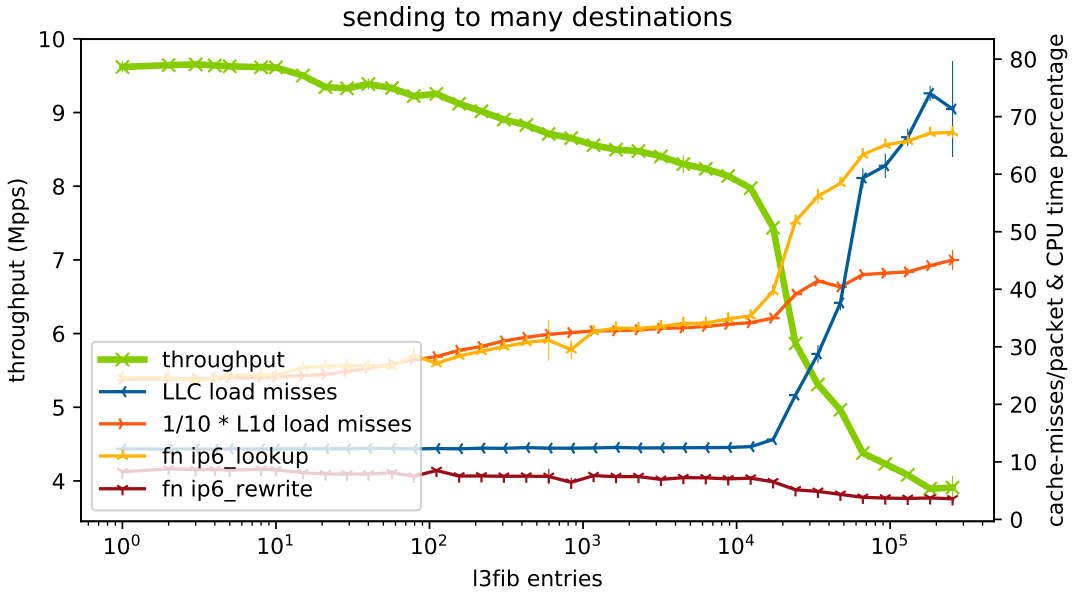

Figure 6.3: Testing VPP v18.10 with different IPv6 fib sizes. Throughput, Layer 1 data cache load misses (divided by 10) and Last Level Cache load misses per packet and the percentage of CPU time spent in selected functions.

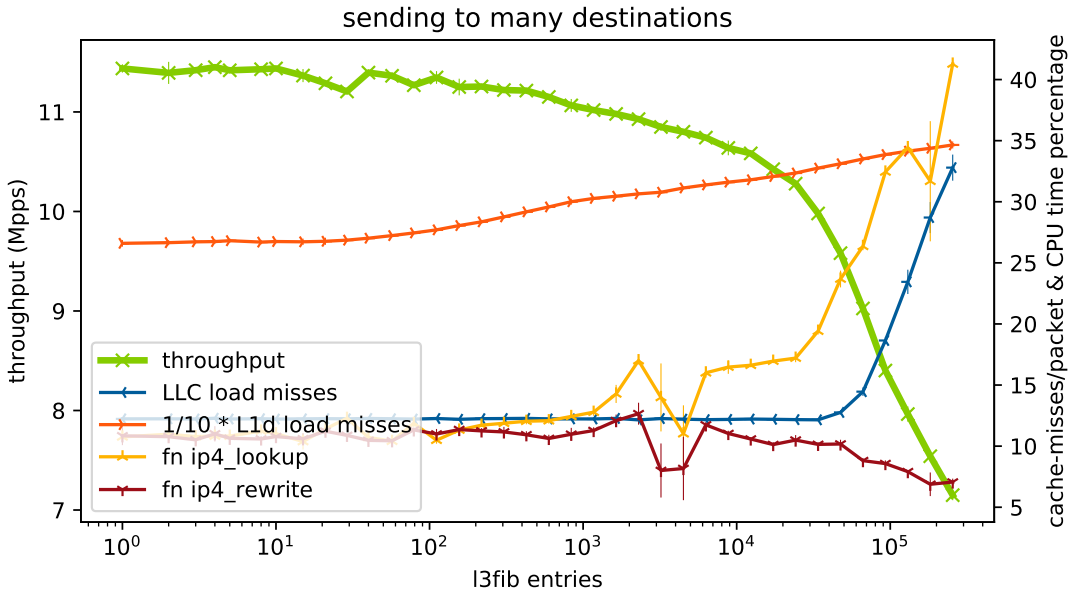

Figure 6.4: Testing VPP v18.10 with different IPv4 fib sizes. Throughput, Layer 1 data cache load misses (divided by 10) and Last Level Cache load misses per packet and the percentage of CPU time spent in selected functions.

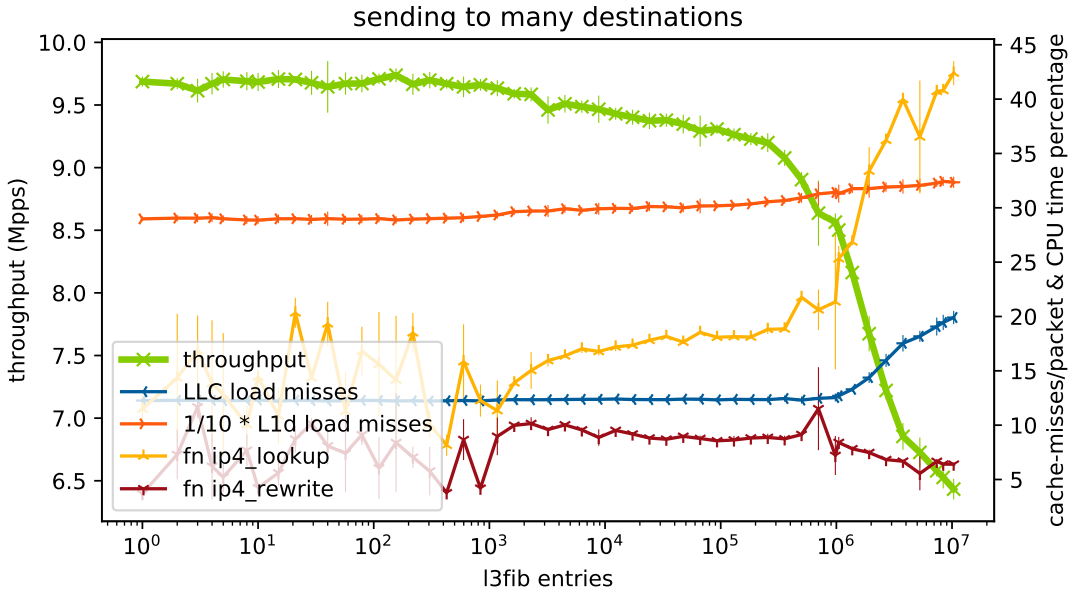

Figure 6.5: Testing VPP v16.09 with different IPv4 fib sizes. Throughput, Layer 1 data cache load misses (divided by 10) and Last Level Cache load misses per packet and the percentage of CPU time spent in selected functions.

The qualitative latency behavior described in figure 6.6 between 1 and 255k routes is similar. The following function can approximate the average latency for different packet rates $t \in (0, t_{max})$ for a known maximum packet throughput t_{max}.

$$l(t) = 59 - 62 * (-\frac{t}{t_{max}} + 1)^{\frac{t_{max}}{8t}}$$

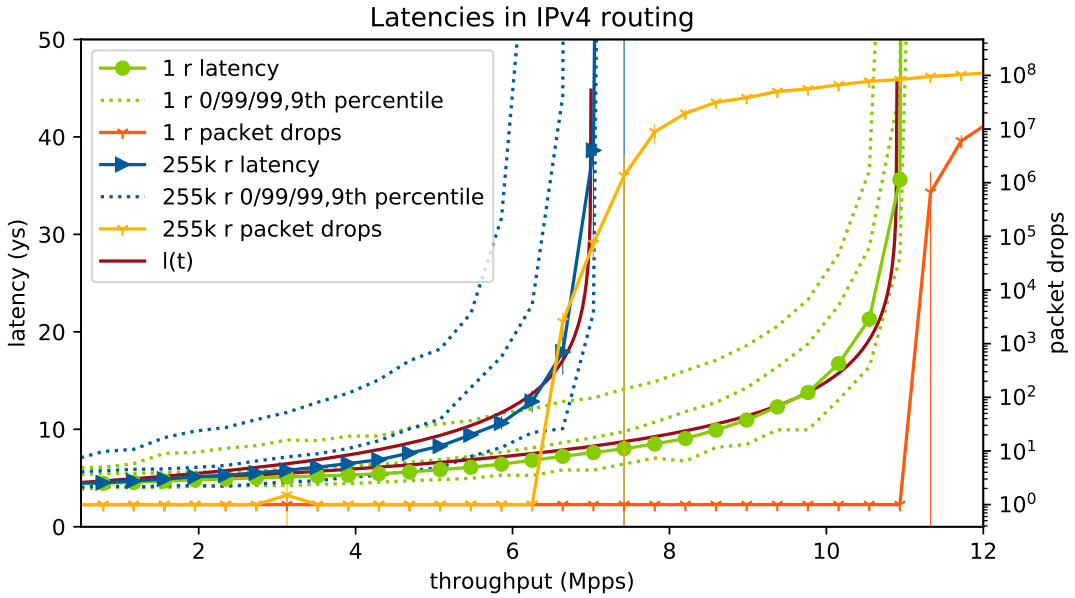

Figure 6.6: Latencies in μs and packet drops of IPv4 routing with a single route (1 r) and 255k routes (255k r).

6.4.2 Latency Distribution

Figure 6.7 shows histograms of latency for IPv4 routing at approximately 10%, 50% and 90% of the maximum throughput rate. It shows that under very low load the latency distribution peaks in the beginning. Later on, the distribution can be approximated as a standard distribution but with growing average and standard deviation. The reason for this is lies within VPP closing the batches as long as the packet processing is not stressed to it's limits (see 4.6). This results in earlier processing of packets at low loads and explains why at 50% of the maximum load the latency is the same, even though the lookup load is different. [14]

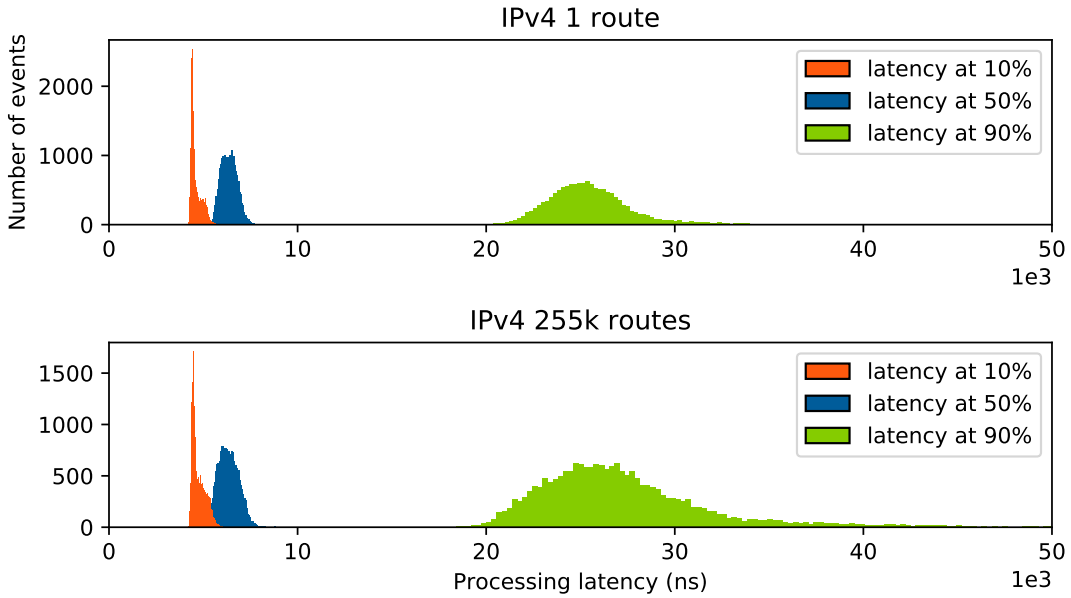

Figure 6.7: Latency histogram for IPv4 routing with a single route and 255k routes at approximately 10%, 50% and 90% of the maximum throughput rate.

6.5 Comparison

Tests showed that there are big performance differences even between VPP versions. While v16.09 is slower when using little IPv4 routes, it doesn't have the fib size limit of 255k which is exceptionally low compared to other software routers. Generally speaking though, it is at least twice as fast as Click DPDK and has similar performance to FastClick DPDK. VPP v18.10 being 1.2 Mpps faster than FastClick DPDK indicates better optimizations and higher potential for VPP v18.10 - but for routing tables with 30,000 to 1,000,000 entries FastClick DPDK has a very clear lead.

Implementation	FIB sizes	Mpps	Relative
MoonRoute	1	14.6	100%
MoonRoute	2^{20}	14.2	97%
MoonRoute	2^{24}	11.6	79%
VPP v18.10	1	11.6	79%
FastClick DPDK	1	10.4	72%
FastClick DPDK	2^{20}	10.4	72%
VPP v16.09	1	9.7	71%
VPP v16.09	255k	9.2	63%
VPP v16.09	2^{20}	8.5	58%
VPP v18.10	255k	7.2	50%
VPP v16.09	2^{23}	6.5	45%
Click DPDK	1	4.3	29%
Click DPDK	2^{20}	4.2	28%
Linux 3.7	1	1.5	11%

Table 6.2: Comparison of maximum IPv4 forwarding throughput with a single worker on the Xeon E3-1230 system (see table 5.1). Non-VPP results are from [9] and are conducted on the same system.

Chapter 7

Conclusion

FastClick and VPP are both fast software routers which vectorize packets during processing and are close to each other performance wise. MoonRoute on the other hand has a significant performance advantage over both of them, being 20-30% faster at routing.

While the performance of VPP v18.10 starts dropping at around 20 fib entries, VPP v16.09 can hold it's highest performance with up to 200 IPv4 fib entries.

Reaching the maximum number of entries, VPP's throughput nearly halves with VPP v18.10 only supporting up to around 287,000 IPv4 fib entries. For this number of routing table entries it is remarkably slower than FastClick with 2^{20} entries, even though it is slightly faster with little table entries.

The best advantage of VPP over it's competitors is it's feature richness. It's during runtime configurable packet processing graph offers for example different tunneling protocols. Settings allow to move the main thread to a dedicated CPU core which in turn allows live inserts of 255k routing table entries with a temporary throughput impact of less than a percent.

Although VPP performs badly with very big routing tables, it's modularity in combination with the rich options to connect it to virtual machines, containers or local high performance applications, make it a well choice for building virtual networks for highly virtualized environments or implementation of Virutal Network Function (VNF).

Next research steps could include more specific benchmarks regarding behavior on receiving control packets or IPv6 specific benchmarking methodology like RFC 5180 [22]. Furthermore the performance change over different VPP versions can be analyzed closer by testing a version between 16.09 and 18.10 and the latest v19.01 which was just re-

leased during the creation of this paper. Especially the code changes leading to the performance differences and how for example FastClick achieves it's high performance with many routes is of interest.

Chapter 8

List of acronyms

CLI	Command Line Interface.
config	Configuration File. Referres to VPP's startup configuration file.
DPDK	The Data Plane Development Kit. `https://www.dpdk.org/`
DUT	Device under Test.
exec	Execution file. Referres to VPP's on startup executed file.
FD.io	The Fast Data Project.
fib	Forwarding Information Base.
HQoS	Hirarchical Quality of Service. A hirachical type of scheduling of packet sending.
IPv4	Internet Protocol version 4.
IPv6	Internet Protocol version 6.
ISO	International Organization for Standardization.
LISP	Locator ID Separation Protocol.
LoadGen	Load Generator.
MAC	Medium access control.
NDR	No Drop Rate. The maximum throughput which doesn't result in packet drops.
NIC	Network Interface Controller.
OSI	Open Systems Interconnection. Reference model for layered network architectures by the OSI.

PDU	Protocol data unit. Refers to a message at a specific layer of the OSI model including all headers and trailers of the respective layer and all layers above.
perf	Linux Performance Tools. [16].
pos	Plain Orchestrating Service. An testbed orchestration tool. [11]
RAM	Random Access Memory.
RSS	Recieve Side Scaling. A DPDK feature for multicore packet processing with hardware acceleration support.
SCTP	SCTP. Datagram-oriented, semi-reliable transport layer protocol.
SDN	Software Defined Network.
SDU	Service data unit. Refers to the payload of a message at a specific layer of the OSI model excluding all headers and trailers of the respective layer.
TCP	Transmission control protocol. Stream-oriented, reliable, transport layer protocol.
UDP	User Datagram Protocol. Datagram-oriented, unreliable transport layer protocol.
VMDC	Virtual Multiservice Data Center. Cisco VMDC 1.x-4.x: SDN and cloud native product groups.
VNF	Virutal Network Function.
vPE	Vritual Provider Edge. A Cisco SDN procuct group.
VPP	Vector Packet Processing. A fast software router.
VXLAN	Virtual Extensible LAN. A protocol for layer 2 ethernet packet encapsulation. Allows gateways to create over 16 million ip-tunnels for layer 2 traffic to a single remote network node.

Bibliography

[1] *2017 BGP Table Size Prediction and Potential Impact on Stability of Global Internet Infrastructure*. http://www.bgphelp.com/2017/01/01/bgpsize/. Accessed on 2019-01-16.

[2] Tom Barbette, Cyril Soldani, and Laurent Mathy. "Fast userspace packet processing". In: *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE. 2015, pp. 5–16.

[3] *BiHash*. https://wiki.fd.io/view/VPP/Bihash. Accessed on 2019-01-16.

[4] Nicolas Bouthors and Christophe Fontaine. *Boosting VNF with Vector Packet Processing*. Aug. 2016. URL: https://www.slideshare.net/QosmosResources/boosting-vnf-performance-with-vector-packet-processing-vpp.

[5] *Configure VPP and run tests on it.* https://github.com/pogobanane/vpp-testing.

[6] Florin Coras and Dave Barach. *VPP Host Stack - Transport and Session Layers*. Speaker Florian Coras at CubeCon CloudNativeCon, Copenhagen, Denmark. May 2018. URL: https://www.youtube.com/watch?v=E9QmvpyeDcE.

[7] Paul Emmerich et al. "Assessing soft-and hardware bottlenecks in PC-based packet forwarding systems". In: *ICN 2015* (2015), p. 90.

[8] Paul Emmerich et al. "Moongen: A scriptable high-speed packet generator". In: *Proceedings of the 2015 Internet Measurement Conference*. ACM. 2015, pp. 275–287.

[9] Sebastian Gallenmüller et al. "Architectures for Fast and Flexible Software Routers". In: ().

[10] Sebastian Gallenmüller et al. "Comparison of frameworks for high-performance packet IO". In: (2015), pp. 29–38.

[11] Sebastian Gallenmüller et al. "High-Performance Packet Processing and Measurements (Invited Paper)". In: *10th International Conference on Communication Systems & Networks (COMSNETS 2018)*. Bangalore, India, Jan. 2018.

[12] Michelle Holley. *Development, test, and characterization of MEC platforms with Teranium and Dronava*. Mobile edge computing delivering cloud computing. Feb. 2018. URL: https://www.slideshare.net/MichelleHolley1/development-test-and-characterization-of-mec-platforms-with-teranium-and-dronava.

[13] "Impressive Packet Processing Performance Enables Greater Workload Consolidation". In: *Intel Solution Brief* (2013). Intel Corporation, Whitepaper.

[14] Leonardo Linguaglossa et al. "High-speed Software Data Plane via Vectorized Packet Processing (Extended Version)". In: *Tech. Rep.* (2017).

[15] *OpenDaylight Lisp Flow Mapping: Architecture*. https://wiki.opendaylight.org/view/OpenDaylight_Lisp_Flow_Mapping:Architecture. Accessed on 2019-01-16.

[16] *perf: Linux profiling with performance counters*. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed on 2019-01-16.

[17] Daniel Raumer et al. "Performance exploration of software-based packet processing systems". In: *Leistungs-, Zuverlässigkeits-und Verlässlichkeitsbewertung von Kommunikationsnetzen und verteilten Systemen* 8 (2015).

[18] Daniel Raumer et al. *Revisiting Benchmarking Methodology for Interconnect Devices*. https://www.net.in.tum.de/fileadmin/bibtex/publications/papers/anrw16-final12.pdf. 2016.

[19] *Release Plan 18.10*. https://wiki.fd.io/view/Projects/vpp/Release_Plans/Release_Plan_18.10. Accessed on 2019-01-16.

[20] *RFC 1242*. https://ietf.org/rfc/rfc1242.txt. Accessed on 2019-01-16.

[21] *RFC 2544*. https://ietf.org/rfc/rfc2544.txt. Accessed on 2019-01-16.

[22] *RFC 5180*. https://ietf.org/rfc/rfc5180.txt. Accessed on 2019-01-16.

[23] *Software Defined Networking*. https://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Data_Center/VMDC/SDN/SDN.html. Accessed on 2019-01-16.

[24] *Vector Packet Processing*. https://docs.fd.io/vpp/18.10/. Official Documentation. Accessed on 2019-01-16.

[25] *Vector Packet Processing (VPP): Integration with Other Systems*. https://fd.io/technology/. Accessed on 2019-01-16.

[26] *VPP: QoS Hierarchical Scheduler*. https://docs.fd.io/vpp/16.12/qos_doc.html. Accessed on 2019-01-16.

[27] *VPP: set dpdk interface hqos placement*. https://docs.fd.io/vpp/18.10/clicmd_src_plugins_dpdk_device.html. Accessed on 2019-01-16.

[28] *VPP: set interface rx-mode*. https://docs.fd.io/vpp/18.10/clicmd_src_vnet.html. Accessed on 2019-01-16.