Technische Universität München

Department of Informatics

Interdisciplinary Project in Electrical Engineering

Performance Analysis of SnabbSwitch

Karthik Mathiazhagan

TECHNISCHE UNIVERSITÄT MÜNCHEN
DEPARTMENT OF INFORMATICS

INTERDISCIPLINARY PROJECT IN ELECTRICAL ENGINEERING

Performance Analysis of SnabbSwitch

Performance Analyse von SnabbSwitch

Author	Karthik Mathiazhagan
Supervisor	Prof. Dr.-Ing. Georg Carle
Advisor	Paul Emmerich, Sebastian Gallenmüller
Date	November 13, 2016

Informatik VIII
Chair for Network Architectures and Services

Abstract

With increasing packet rates offered by modern commodity 10 Gbps Ethernet NIC, the performance of network stacks provided by current operating system is decreasing and can no longer match with these packet rates. This bottleneck of network stacks opened a domain for development of high speed packet processing frameworks which can outperform the performance of traditional network stack. Deployment of these high speed packet processing frameworks on commodity hardware together with flexible software are the current trend in today's networking domain.

Snabbswitch or Snabb is one such open source high speed packet processing framework developed by Luke Gorrie since 2012. It combines several novel methods to achieve the performance required to cope up with packet rates offered by 10 Gbps Ethernet NIC which was previously only possible by dedicated hardware networking devices. In this report we present the basic architecture of Snabb, usage of its high level API's to construct a network function, understand some of its built in applications and finally analyse performance of the framework. Later we discuss how Snabb application are written, linked together as a application chain and executed. We then present several performance characteristics such as throughput, latency and cache events of Snabb apps which then could be assessed before using this framework for deploying network functions.

Contents

List of Figures

Chapter 1

Introduction

Snabb [2] is a simple and fast packet processing tool kit. The project was started by Luke Gorrie in 2012 and since then its in constant development cycle. The aim of the project was to provide network administrators, internet service providers a software solution to cope up with processing huge date rates offered by modern commodity 10 Gbps Ethernet NIC rather than dedicated hardware solution. Snabb combines several novel methods to achieve this functionality such as,

- Developed using Lua programming language and uses LuaJIT compiler.
- User space networking by kernel Bypass I/O [10].
- Virtualization using SnabbNFV module.

Network function virtualization (NFV) [1] is currently the trend in today's networking domain. NFV offers a new way to deploy and manage networking services. NFV allows to create Virtual Network Functions (VNFs) for implementing networking functionalities such as network address translation (NAT), firewall, domain name systems (DNS), routing and much more. These VNF are software solution which run inside commodity x86 hardware rather than on dedicated proprietary hardware. Some of the key objectives of NFV technology is to provide improved capital efficiency and improved flexibility. This is achieved by implementing the network function on software running above the virtualization layer thereby decoupling the functionality from the underlying hardware architecture. Network function virtualization implements virtual network functions as software only entities that run on top of a so called Network function virtualization infrastructure (NFVI). One of the practical implementation of network function virtualization was announced by Deutsche Telecom in its Terastream project [5]. The project aims at replacing their backbone infrastructure to a cloud based IP network architecture. This way the backbone network would be native IPv6 network while offering services to IPv4 consumers. The solution would be to implement virtual network functions servicing both IPv4 and IPv6 networks from inside of a virtual machine rather than running on proprietary hardware. The virtualization layer on top of which network function are implemented should support the requirements of NFV in terms of efficiency, performance and security. Snabb framework has developed a dedicated module called SnabbNFV [23] to support implementation of network functions on top of the virtualization layer satisfying the above mentioned requirements. Later in this document we would discuss about one of Snabb apps called Lightweight 4over6 [24] "lwaftr" which is being used by Terastream project for IPv4-over-IPv6 networks.

Snabb is build along any modern Linux operating system and can be compiled into a standalone executable 'snabb'. This executable then could be executed on any modern Linux/x86-64 distribution. Software frameworks such as netmap [26], Intel DPDK [21], PF_RING ZC [13] are also similar frameworks for the purpose of high speed packet I/O. This report focuses on Snabb and its performance characteristics, as the rest of the frameworks [8] has been analysed before.

This report is structured as follows. In Chapter 2 we discuss about the basic architecture of Snabb, its core modules and we explore how to write a sample Snabb forwarder application. We will also discuss about network function virtualization handled by Snabb using vhost-User approach. In Chapter 3 we evaluate the performance of Snabb by briefly analysing the forwarder application. Also we would evaluate 'lwaftr' module of Lightweight 4over6 to assess its performance. Finally in Chapter 4 we use our results obtained before to summarize and conclude.

Chapter 2

Snabb

2.1 Snabb

2.1.1 High Level Design

Snabb is comprised of core modules and high level APIs which allows users to design their own network functionality. Since Snabb is almost written in Lua programming language, designing a network function is all about writing a Lua script. Lua is a light weight scripting language which is easy to learn and program. LuaJIT [14] is a trace based just-in-time compiler [15] for Lua which can optimize the code behaviour based on traces obtained during run time.

Also the simple Foreign Function Interface (FFI) library of LuaJIT allows it to interact with low level system APIs and data structures written in C. FFI library is tightly integrated with LuaJIT and should be loaded to access C data structures. Using FFI library can drastically improve the performance of the program as compared to pure Lua implementations. Another huge gain is the memory of the program, FFI implementations could make use of C structures which would occupy lesser memory as compared to Lua tables. Further in order to get the full control of the network interface, Snabb has written its own device drivers [11] in Lua for modern commodity Intel NICs. Due to the above reasons, Lua and LuaJIT were chosen as programming language rather than traditional C or C++.

2.1.2 Snabb Core Modules

Every network design in Snabb can be realised as an application or app. Each app has its own functionality and can be connected with rest of the apps to form a chain of app network. Every app has one or more than one input and output links which allows them to be linked with the app network. The basic functionality of any app would be to pump the traffic from input link, do the processing as scripted and push the traffic to the output link. Links are implemented as circular ring buffers [16] which is used to pass traffic between one app to another. Snabb also comes with built in apps and libraries which implements certain basic network function. This eases the developer to just focus on implementing the required network functionality alone.

Every Snabb application has a configuration, which is comprised of list of apps and the links between them. It can be thought of as a directed graph with nodes as apps and links as circular ring buffers between them. Finally, Snabb engine takes in the configuration as input, constructs the app structure, initializes the circular buffers between the apps and starts executing the configuration.

Snabb engine has so called breath cycle, which consist of inhale and exhale cycle. During each breath cycle the so called 'push' and 'pull' routines of every app are executed. The basic idea is that, 'pull' routine pulls the packet from the input link of the app and make the packet available for processing and 'push' pushes the packet from the app to its output link. This way every Snabb app breaths by pulling, processing and pushing the packets.

2.1.3 Snabb Forwarder App

The previous sections gives us a brief understanding of Snabb core modules. In this section we would look on how to write a basic Forwarder app. The purpose of this forwarder is to demonstrate how small and simple apps could be linked together to build a complex app network implementing a network functionality. As the name suggest Forwarder app forwards the packet from one Intel82599 NIC to another Intel82599 NIC. As stated earlier, Snabb has its own driver code written in Lua for the underlying Network Interface Controllers. The following are the devices supported by Snabb currently,

- Intel 82599 SFP
- Intel 82574L
- Intel 82571
- Intel 82599 T3
- Intel X540
- Intel X520
- Intel 350
- Intel 210
- Solar flare SFN7122F

The forwarder could be configured either as unidirectional or bidirectional with varying number of apps in the app chain. These parameters would be supplied as arguments during initialization of the app. The processing part of each app in the forwarding chain would be simply to receive the packets from the input link and forward it back to the output link. To better illustrate, listing 2.1 shows the 'push' and 'process_packet' routine of each forwarder app.

Listing 2.1: Forwarder Push and process_packet routine

```

function Simpleforwarder:push()
   local i = assert(self.input.input, "input port not found")
   local o = assert(self.output.output, "output port not found")

   while not link.empty(i) and not link.full(o) do
      self:process_packet(i, o)
      self.packet_counter = self.packet_counter + 1
   end

   if link.full(o) then
        print("Output link is full")
   end
end


function Simpleforwarder:process_packet(i, o)
   local p = link.receive(i)
   link.transmit(o, p)
end
```

As we can see that 'process_packet' just receives a packet from input link and transmits it to output link. This describes the individual processing element of the app. As stated in previous section, the configuration for this forwarder constructs the chain of these apps depending upon the 'noofhops' command line argument. Finally to get hold of the underlying network device receive and transmit queues, Snabb has built in 'Intel82599' app which finishes our app chain. The 'tx' and 'rx' links of 'Intel82599' app are memory mapped regions [25] of transmission and receive queues of the NIC device. Packets received from the NIC are put into the 'tx' link and packets from the app network are put into 'rx' link. Listing 2.2 shows how chain of forwarding apps are built along with 'Intel82599'.

Listing 2.2: Forwarder app construction

```
function run (parameters)
   if not (#parameters == 4) then
      print("Usage: forwarder <input PCI Bus:Device.function>
      <output PCI Bus:Device.function> <No Of Forwarding apps>
       <Direction 'B' or 'S'")
      main.exit(1)
   end
   ...
   print("Configuring app network ...")
   config.app(c, defaultforwarderfwdstart, Simpleforwarder.Simpleforwarder)

```

```
12    local Interfaceapp = "ioIntel82599"
13        local assigned = 0
14        for _,device in ipairs(pci.devices) do
15           if is_device_suitable(device, inputportpattern) then
16              config.app(c, Interfaceapp, Intel82599, { pciaddr=device.pciaddress})
17              print("Linking app "..Interfaceapp.." and "..defaultforwarderfwdstart)
18              config.link(c,Interfaceapp..".tx -> "..defaultforwarderfwdstart..".input")
19              assigned = 1
20           end
21        end
22        assert(assigned >0," PCI Device does not match any suitable device")
23
24        BuildandLinkInterface(c,defaultforwarderfwdstart,Interfaceapp,NoOfHops,'Frwd_To')
25        ...
26        engine.configure(c)
27     engine.main()
28 end
```

Lines 10 and 16 shows initialization of 'Simpleforwarder' and 'Intel82599' app and line 17 is where the link 'tx' of Intel82599 is linked to 'input' of 'Simpleforwarder'. This makes them point to the same circular ring buffer. To finish the set-up, 'BuildandLinkInterface' builds the chain of 'Simpleforwarder' app and finally links 'output' of 'Simpleforwarder' to 'rx' of 'Intel82599'. Snabb Engine then takes the configuration 'c' as input and configures the app chain and finally executes them by calling the 'main' routine. Once the configuration is validated and initialized the breath cycle of Snabb Engine starts pumping the traffic through the app network.

These simple and flexible design of Snabb allows user to make use of its API easily to start constructing a network function. Built in custom apps and libraries help us to manipulate the header, data sections of a packet, construction/cloning of packets and adds much more functionality. To finish this section, the following Figure 2.1 shows the structure Bi-directional forwarder app with total of 2 apps in the network. Each of 'Intel82599' app drives the underlying NIC and is connected to the forwarder apps 'Fwrder 1' and 'Fwrder 2' via 'tx' and 'rx' links. MoonGen [6,7] in the diagram is a high speed packet generator tool which we used as source for the test traffic. MoonGen was used to send packets on both 'ethtest0' and 'ethtest1' links and receive them again in opposite directions.

2.2 Snabb NFV with Vhost-User

In previous section we discussed briefly about Snabb architecture its core module and also discussed about writing a basic Snabb app. In this section we shall discuss about Snabb app 'snabbnfv' to support Network Function Virtualization combined with vhost-user feature of QEMU that was developed by Virtual Open Systems [19].

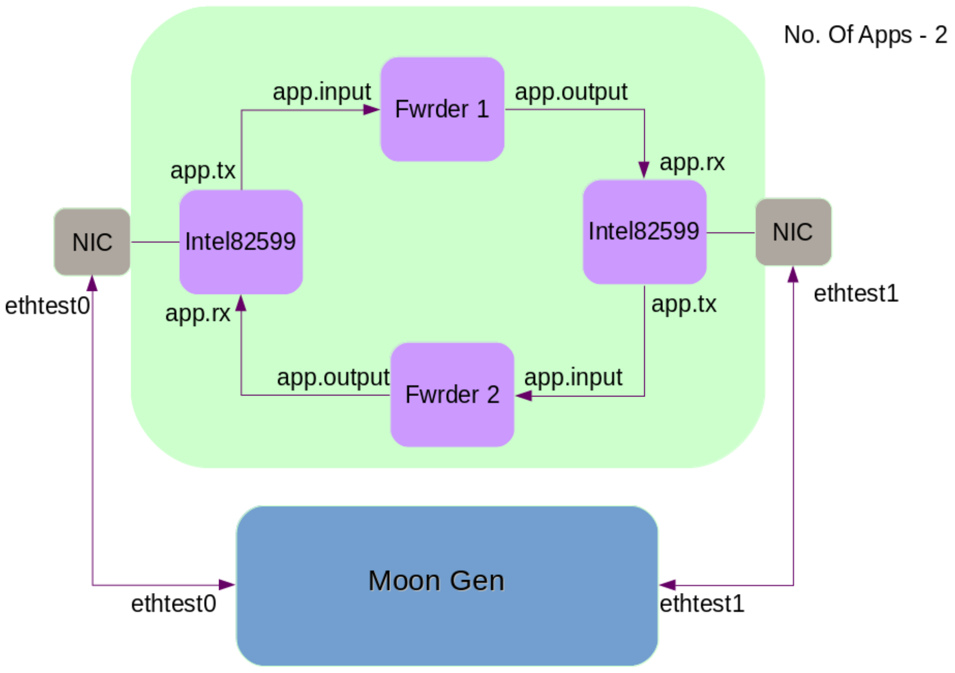


Figure 2.1: Structure of Bi-Directional Forwarder App

2.2.1 Virtio Infrastructure with Vhost-User

Virtual Machines created by QEMU/KVM may acquire network functionalities by virtio_pci devices. This virtio_pci device is a para-virtualized device where its drivers are divided into two parts 'front-end' and 'back-end' drivers. Generally, the 'front-end' driver part is implemented in the virtual machine and 'back-end' drivers are implemented by the para-virtualization layer which is QEMU in our case. Communication between the 'front-end' and 'back-end' drivers is established by means of virtio queues which are implemented as virtio ring data structures. In a virtio infrastructure the QEMU emulated virtio_pci device implements the virtio ring data structure which servers as a standard mechanism to implement the virtio drivers utilizing this virtio ring data structures. In other words, the virtio_pci device emulated by QEMU serves as a transport mechanism between the virtual machine 'front-end' driver and QEMU 'back end' driver as they now share the virtual queues.

One important thing to notice here is the network traffic need to be process by QEMU layer before passing it to the host kernel network stack and also the other way. The solution to this problem was 'vhost-net' [18]. This solution allows the user space process (i.e QEMU VM) to share a number of virt queues directly to the kernel driver. This way network traffic from host can directly reach the guest VM without interference of QEMU thereby significantly improving the performance. In a virtio infrastructure with 'vhost-net' QEMU will still emulate the virtio_pci device and have a hold of its control

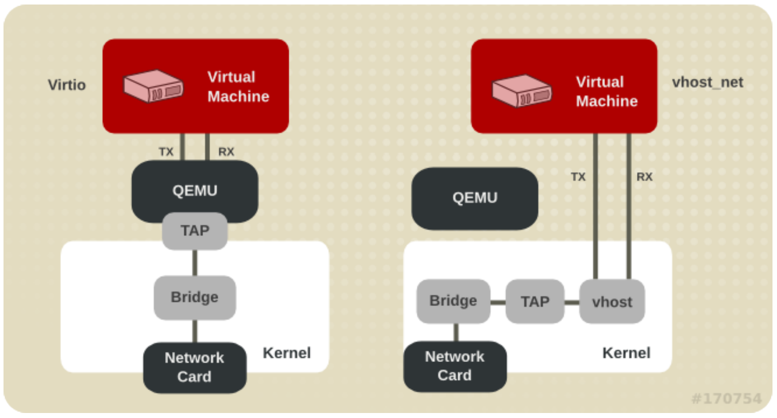


Figure 2.2: QEMU Virtio pci device using vhost-net

plane. But once the device is emulated its data plane which is the virt queues are now shared with host kernel with the help of vhost api. ’vhost-net’ driver would now bypass the QEMU layer and pass the traffic from guest virtual machine to a tun/tap device in the host kernel. This tun/tap device can now be attached to a kernel network bridge to pass the traffic to outside world. Figure 2.2 taken from red hat customer portal [18] gives us a good understanding of ’vhost-net’ bypassing QEMU.

The above solution of ’vhost-net’ would still not be an ideal solution for Snabb as Snabb is completely running in user space. In a Snabb environment the network traffic does not traverse the kernel space but it is immediately available in userspace from the hardware pci device. To achieve the performance required by Snabb ’vhost-net’ would not be a ideal solution and therefore is replaced by ’vhost-user’. With qemu2.1 release it supports ’vhost-user’, the goal of ’vhost-user’ is to allow the virtio queues be directly shared between another user space process running the virtio-net back end driver and also to preserve the vhost paradigm of using shared memory and event triggers using ’ioeventfds’ and ’irqfds’. This way network traffic from one user space process implementing the virtio-net front end driver can now be directly passed to another user space process implementing vitio-net back end driver. Figure 2.3 taken from virtual open systems [17] gives us a realization of direct Snabb to QEMU guest virtio-net communication. The ’vhost-user’ addition into QEMU gives us additional options to configure the interface for communication between the vhost-user front-end and back-end. These options includes a ’mem-path’ parameter which allows the Guest RAM as a shared memory space between the two user space process, a UNIX domain socket for communication between QEMU and user space vhost implementation, and finally the user space vhost implementation will receive the file descriptors of Guest shared RAM thereby it can directly access the virtio rings of Guest.

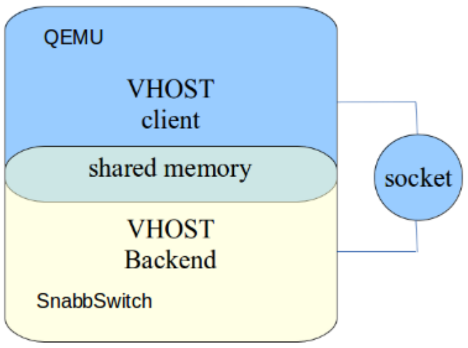


Figure 2.3: Vhost-User Architecture

2.2.2 Snabbnfv with Vhost-User

In this subsection we shall discuss about configuring 'snabbnfv' and to pass traffic to QEMU Guest using vhost-user approach discussed in previous section. 'snabbnfv' is an app implementing the network function virtualization paradigm for Snabb infrastructure. It is typically deployed for Open Stack Neutron with components for Compute Node, Database Node and Network Node. However in this section we shall discuss about configuring it to pass traffic to Guest VMs. Snabb NFV takes in a configuration file, pci address and a socket path as input and configures a app network which would finally be executed by Snabb engine. The pci address would be used by the Snabb network driver, socket path would be used by the vhost-user back end implementation by Snabb and configuration file would be used to construct the app network. The steps we followed to run 'snabbnfv' using vhost-user is provided in listing 2.3.

Listing 2.3: Running Snabbnfv with vhost-user

```
# Step 1: Create and Mount Hugepages. This hugepages would
# be shared memory region between QEMU Guest and Snabb
# vhost-user app.

# Edit /etc/sysctl.conf file to add appropriate number of
# hugepages

vm.nr_hugepages = 12288

# Create mounting directory and mount

sudo mkdir /mnt/huge
sudo mount -t hugetlbfs nodev /mnt/huge

# Step 2: Create a Configuration file
# mac_address is the mac address of the QEMU Guest to pass
# the traffic and port_id is used to identity the socket name

---ports1.cfg---
return {
  { mac_address = "52:54:00:10:10:10",
      port_id = "id1",
  },
}

# Step 3: Create a directory where QEMU sockets would be
# created and attached by Snabb vhost-user

mkdir /root/vhost-user

# Step 4: Run snabbnfv traffic process

sudo ./src/snabb snabbnfv traffic -k 10 -D 0 \
  0000:02:00.0 ./port1.cfg /root/vhost-user/vm1.socket

# Step 5: Start the QEMU/KVM Virtual machine

sudo /usr/local/bin/qemu-system-x86_64 -curses -name palanga-vm1
 -drive if=virtio,file=/root/vm/grml-3.16-xen/initrd.img -M pc -smp 2
 --enable-kvm -cpu host -m 8192 -numa node,memdev=mem
 -fsdev local,security_model=passthrough,id=fsdev0,path=/root/qemu/snabb
 -device virtio-9p-pci,id=fs0,fsdev=fsdev0,mount_tag=share
 -object memory-backend-file,id=mem,size=8192M,mem-path=/mnt/huge,share=on
 -chardev socket,id=char0,path=/root/vhost-user/vm1.socket,server
 -netdev type=vhost-user,id=net0,chardev=char0
 -device virtio-net-pci,netdev=net0,mac=52:54:00:10:10:10
 -netdev type=tap,id=net2,ifname=ethvm,vhost=on
 -netdev bridge,id=hostnet0,br=br-mgmt
 -device rtl8139,netdev=net2,mac=52:54:00:00:07:02
```

From the above listing, we could see that in step 4 we run the snabbnfv process giving the configuration file, pci address and a path to a socket as input parameters. Once snabbnfv process runs it would create a vhost-user back end as a server opening the given socket. In step 5, we start the QEMU virtual machine by giving the same socket path as a parameter to vhost-user net device. This will now run vhost-user as a client and attach to the socket created by the server. Note that in step 5 we are creating two virtio network interfaces in the Guest one using 'vhost-user' approach and another using 'vhost-net' approach. The virtio interface created using 'vhost-net' would be used for Ethernet management interface and virtio interface created using 'vhost-user' approach would be used to process huge amounts of network traffic. Also we give the 'mem-path' parameter with the path we created using huge pages. This memory in Guest RAM [20] would be now shared between QEMU Guest user space process and Snabb vhost-user application.

Chapter 3

Performance Evaluation

In this section we briefly evaluate the performance of Snabb by executing 'Simple-Forwarder' app discussed in previous section and also some of built in apps of Snabb. The experiments were conducted on Baltikum test bed servers equipped with Intel(R) Xeon(R) CPU E3-1230 V2 clocked at maximum of 3.30GHz, supermicro X9SCL/X9SCM motherboard with 16GB DDR3 SDRAM and two Intel 82599EB Network Connections. The set up uses two servers connected to each other with 10 Gbps Ethernet links. One server runs Snabb and another runs the packet generator tool which is the source of test traffic. Figure 3.1 shows us the basic set up.

We used MoonGen [6], which is a high speed flexible packet generator as the source for test traffic. MoonGen can send 14.88 Mpps, line rate at 10 GbE with minimum-sized packets, from a single CPU core. Another significant feature of MoonGen is measurement of latency with sub-microsecond precision by using the hardware time stamping capabilities of modern Commodity NICs.

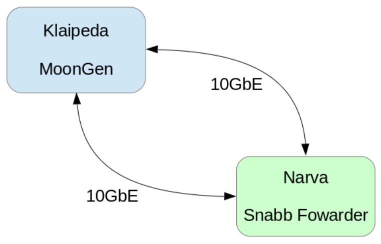
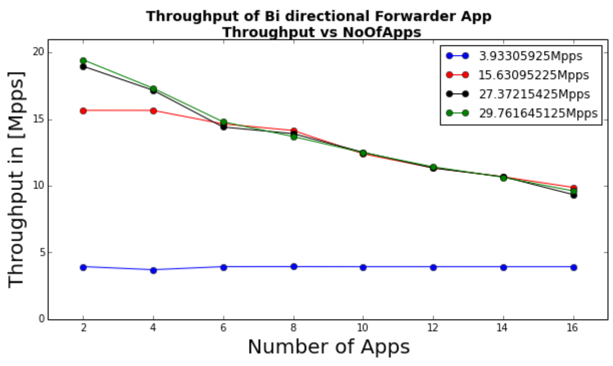
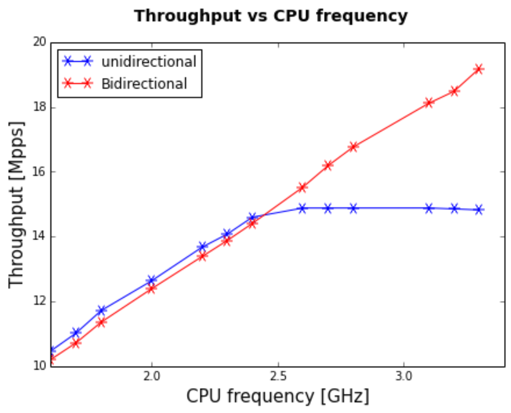


Figure 3.1: Baltikum Test bed setup.



(a) Throughput vs No of apps



(b) Throughput vs CPU frequency

Figure 3.2: Throughput of Bi-Directional Forwarder

3.1 SimpleForwarder

3.1.1 Throughput and Latency Measurements

We measured the maximum throughput of the Forwarder app at different packet rates. The experiment was done using minimum packet size of 64 bytes. Since various factors affects the throughput of the forwarder app, detailed measurements were taken for each of the factors. Throughput is widely affected by the processing speed of the CPU, i.e the number of CPU cycles spent on receiving the packets, processing them and transmitting them. Measurements were taken with increasing CPU frequency which affects the maximum load per CPU. Number of apps on the forwarding chain is another factor which could affect the throughput of the application. Throughput gradually decreases as the numbers of apps in the forwarding chain increases. To capture this, measurements were taken with varying number of apps in the forwarding chain. Finally, packet size was also considered as one of the factor for the measurement.

Figure 3.2a depicts the throughput of Forwarder app for different packet rates in relation

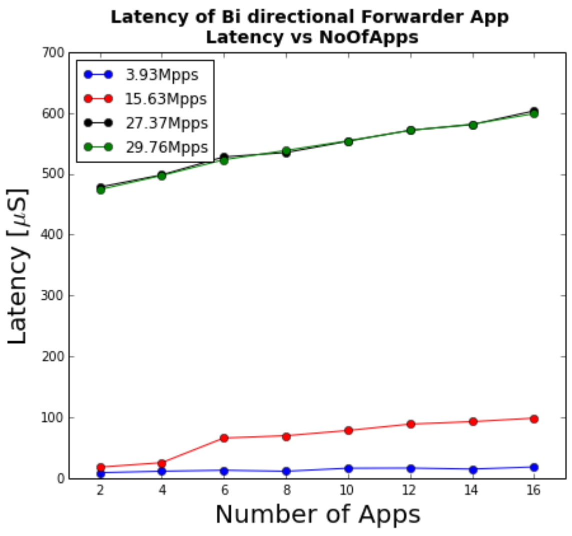


Figure 3.3: Latency of Bi-directional Forwarder.

to the number of apps in the forwarding chain. It could be drawn to conclusion that for lower packet rates throughput almost remains constant even for larger number of apps in forwarding chain and with increase in packet rate the throughput declines gradually. For input packet rate of 15.6 Mpps, throughput starts to decline when number of apps are greater than 4. This could be the threshold packet rate for 4 apps in the forwarding chain. Similarly Figure 3.2b shows the relation between CPU frequency and throughput. Single direction forwarder achieves its maximum throughput of 14.88 Mpps at around 2.6 GHz of CPU frequency and remains constant there after where as Bi-directional forwarder tends to reach its maximum throughput around 19 Mpps at maximum CPU frequency of 3.3 GHz.

Latency measurements were also done as part of the previous experiment. Latency measurements were calculated at the load generator(MoonGen) which uses hardware time stamping capabilities of the NIC [9]. Results were as expected and depicted in Figure 3.3. Average latency tends to remain constant with lower packet rate and starts to shoot up drastically on high packet rates. As with throughput measurement, packet rate of 15.6 Mpps remains to be the threshold rate for latency with 4 apps in the forwarding chain. Beyond which the latency starts to shoot up gradually.

To capture the behaviour of forwarder for packets with different sizes, throughput measurements were repeated with varying packet sizes. MoonGen was again used to generate packets of varying sizes starting from 64 bytes to 1518 bytes. As the packet sizes gradually increases MoonGen showed a reduction in the packet rate as expected. Forwarder was able to forward packets with the same packet rate as MoonGen. Figure 3.4 shows the throughput of single direction forwarding app for different packet sizes. We can see that the throughput has a steep fall from packet size of 64 bytes to 256 bytes,

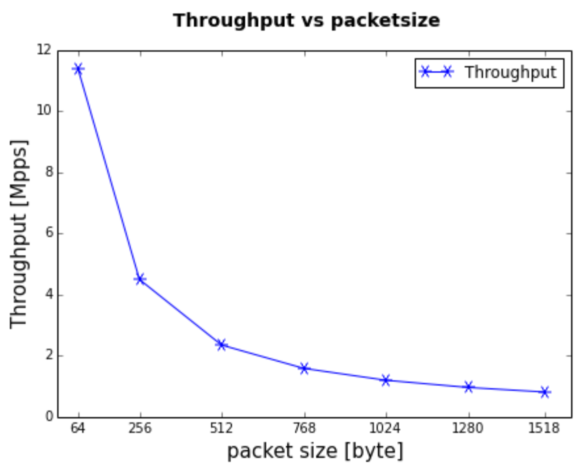


Figure 3.4: Throughput vs Packet size

after which it decreases gradually.

3.1.2 Profiling Forwarder App

LuaJIT traces

LuaJIT is a just-in-time compiler for Lua programming language. LuaJIT combines high speed interpreter and a state-of-the-art-JIT compiler. In contrast to method based JIT compilers which translates one method at a time to machine code during runtime, LuaJIT compiler is a trace base JIT compiler, which looks for program loops as hotspots [12] and compiles them to traces during runtime. Trace based JIT compiler works on assuming that program spends frequently on a particular loop and subsequent loop iteration takes the same path. In those cases it is easy to optimize those execution units. Those optimized execution units during run time are called LuaJIT traces.

When Lua program byte code is in execution, LuaJIT enters into recording phase where recording of a trace happens. During this phase it starts to generate IR (Intermediate Representation) instructions of the trace which is later translated to machine code [15]. A CPU execution could be kept very fast as long as its on a single trace and causes significant overhead when there is jump between traces. Owing to the above reason LuaJIT traces have some characteristics such as, no branches, typically in a single loop and all function calls are inlined.

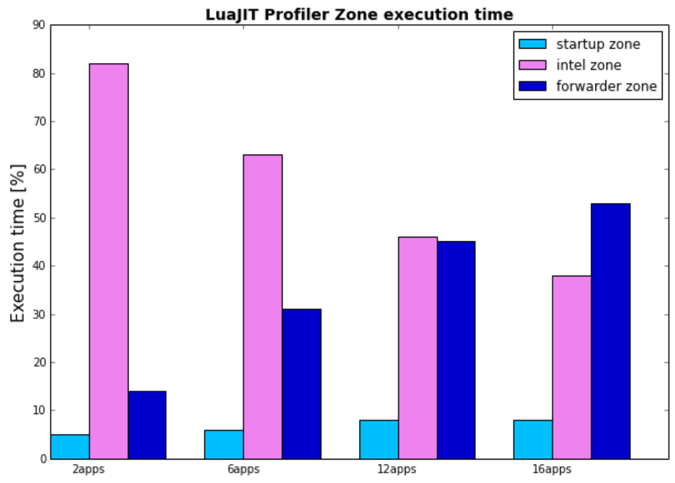


Figure 3.5: LuaJIT Zone execution time

LuaJIT Profiler

LuaJIT 2.1 [14] release features an integrated statistical profiler with a very low overhead. The integrated profiler could be accessed using high level API using (-jp command line option), low level Lua and C API. It could be used to generate stack traces, source code annotations, source code execution time and also trace dumping. One more interesting feature is the usage of LuaJIT profiler to get zone execution time. This could be very useful in analysing which zone of the program occupies most of the time during various scenarios. Lua zones are nothing but information about different parts of the application. For our forwarder applications, there are three major zones namely 'startup' where the app engine responsible for program start up resides, 'intel' zone which occupies device drivers and does receiving and transmitting packets to device part of the program and 'simpleforwarder' zone which runs the forwarder code.

We used LuaJIT profiler to obtain information about CPU execution time for various zones in the forwarder app. The profiler was invoked using '-jp' options supplying '3zFm1' as its parameter value which provides statistical information with 3 stack level deep, zone information, shows module function names with minimum sample percentage of 1. We ran the bi-directional forwarder app, and profiling information for different zones was sampled for 2,6,12 and 16 number of apps in the forwarder chain. Figure 3.5 shows the results summarized as a graph.

We also dumped various traces of forwarder app along with their execution time. Since each zone could have several traces, zone 'intel' had many traces as compared to 'simpleforwarder' zone which had only one trace. The below figure 3.6 shows execution time split up based upon the traces obtained during profiling bi-directional forwarder with 10 apps on forwarding chain. The traces for 'startup' zone was ignored because it

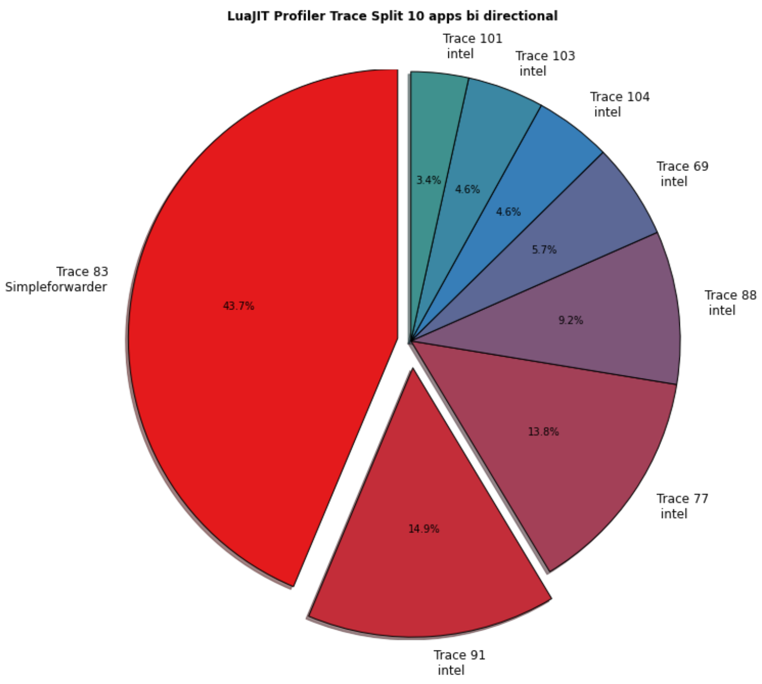


Figure 3.6: LuaJIT Trace execution time

had less than 3% of CPU execution time.

3.2 LightWeight 4over6 lwaftr

One of the interesting Snabb application which demonstrates the application of Network Function Virtualization (NFV) is Snabb 'lwaftr' [24]. Lightweight 4over6 (lw4o6) is IPv6 transition mechanism and Snabb 'lwaftr' is one part of it. It is implemented as per RFC 7596 in which Snabb 'lwaftr' is the internet facing component of lw4o6. This transition mechanism allows network providers to completely migrate to IPv6 while still maintaining interoperability with IPv4 internet to the customers. The basic functionality of Snabb 'lwaftr' is encapsulation and decapsulation of IPv4 over IPv6. Another component of lw4o6 which is present in the customer premises is B4 or 'lwB4'. 'lwB4' and 'lwaftr' together form lightweight 4over6 for IPv6 transition. Each customer IPv6 addresses is mapped to limited range of port numbers in the IPv4 address. This allows the customer site to share the same IPv4 address while maintaining different port ranges. Mapping is done by Snabb lwaftr by maintaining a binding table of B4's IPv6 address, the allocated IPv4 address and the Port range. Due to this simplicity in design and configuration network providers could use Snabb lwaftr to support their backbone network with IPv6 while still servicing IPv4 customers. Figure 3.7 better illustrates lwaftr architecture diagram which was presented by Katerina Barone-Adesi and Andy

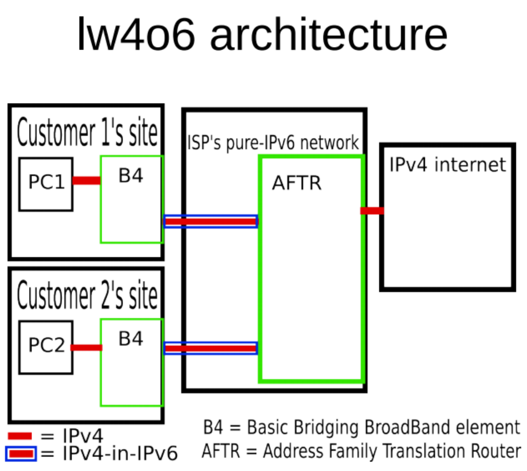


Figure 3.7: Lwaftr architecture

Wingo during FOSDEM 2016 [3, 4].

3.2.1 Encapsulating IPv4 into IPv6

As discussed in previous section, 'lwaftr' uses binding table to associate which 'B4' is responsible for an incoming IPv4 packet. Binding table in simple terms consist of set of softwires(tunnels) in which one end point in 'B4' and another endpoint is 'lwaftr'. Lwaftr at minimum has two interfaces in which one interfaces faces the internet where traffic is IPv4 and another interface communicates with B4 where traffic is IPv6. To get a good understanding, we analysed the internal app network of 'lwaftr' and the links between them during encapsulating a IPv4 packet into an IPv6 packet. We found that there were total of 8 apps chained together performing IPv4 encapsulation. While each app in the chain has its own functionality, input to the app chain starts from "inetNIC" app facing the internet side(IPv4) and final output comes from "b4sideNic" app facing the B4 (IPv6) network. Following figure 3.8 shows the app structure and links between them during Encapsulating IPv4 packets.

3.2.2 Performance

Performance of 'lwaftr' was analysed by running 'loadtest' module of lwaftr which pumps both IPv4 and IPv6 on both the interfaces and also records the received rate. One baltikum test bed server equipped with Intel(R) Xeon(R) CPU E3-1230 V2 clocked at maximum of 3.30GHz with two Intel 82599EB Network Connections was running lwaftr with one interface configured for IPv4 and another interface configured for IPv6 traffic. Another test bed server was running loadtest module pumping IPv4 and IPv6 traffic on respective interfaces. Initial measurements for IPv4 encapsulation were taken

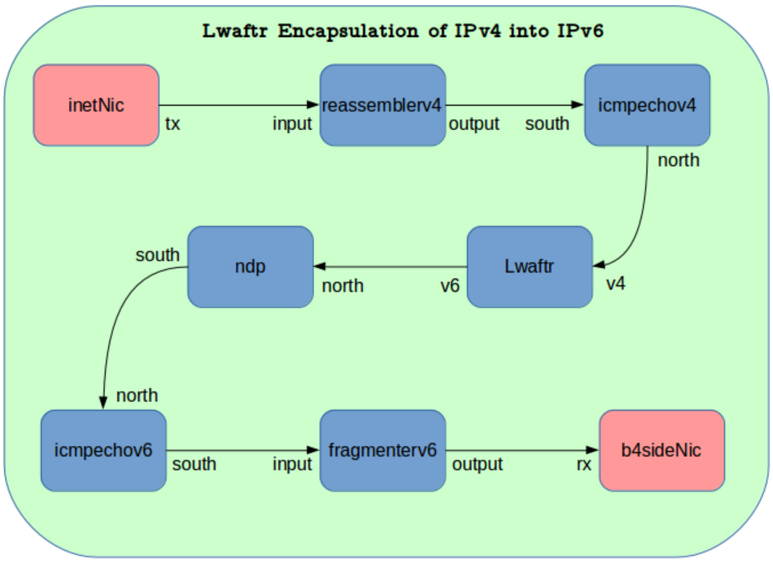


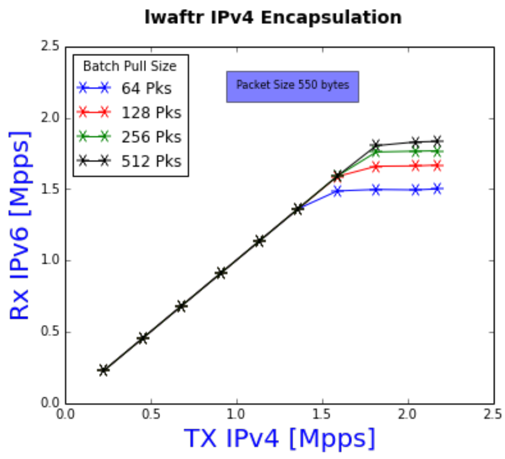
Figure 3.8: Lwaftr encapsulation of IPv4 over IPv6

and found that lwaftr was able to process close to 1.7 Mpps where as the benchmark of lwaftr shows processing up to 2.2 Mpps. Later we found that it was due to usage of large binding table as result of which the packet loss start to shoot up drastically with increase in input packet rate.

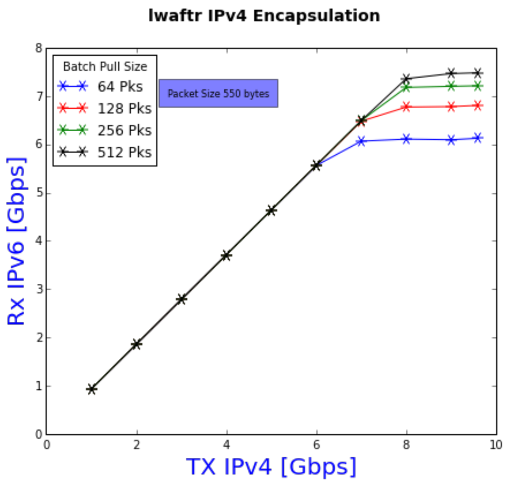
3.2.2.1 Influence on Batch Size

One factor which influences the processing capability of Snabb applications is the Batch pull size of Intel82599 driver. Snabb by default had a batch pull size of 128 packets, which means one call of pull() routine of Intel82599 could pull atmost 128 packets into the app network. We tried to increase this pull size and tried to analyse its performance impact on the application. We found that the application had significant performance improvement when batch pull size was increased to certain limit, but it could not be increased more than the buffer size of the links present between the apps as it resulted in buffer overflows and packet loss. So we made the Batch pull size of Intel82599 configurable by passing it as a runtime parameter to 'lwaftr'. Proper validations were made to ensure that the batch size could not be more than the link buffer size of Snabb.

Encapsulation of IPv4 over IPv6 of 'lwaftr' was again repeated with different batch pull size and the results showed that lwaftr could process close to 2 Mpps IPv4 packets for batch size of 512 packets. Link ring size was kept at 512 packets. We also found that the performance improvement gained by increasing batch size from 128 to 256 packets were large compared to the improvement gained from batch size of 256 to 512 packets. Figure 3.9b shows the processing rates of lwaftr during IPv4 encapsulation for different batch sizes. It could be made clear that configuring the batch pull size could improve the



(a) IPv4 Encapsulation in Mpps



(b) IPv4 Encapsulation in Gbps

Figure 3.9: lwaftr IPv4 Encapsulation for differnet batch size

performance of 'lwaftr'. Also we can see that for the batch pull size of 256 packets which is half the buffer link ring size (512 packets) the performance improvement gained is maximum.

Chapter 4

Conclusion

We presented Snabb which is a high speed packet processing framework, its novel design and architecture. Snabb is able to overcome the problems faced by traditional network stack and is able to handle huge packet rates that was only previously possible by dedicated hardware solutions. Its kernel by-pass technology is needless to say provides the key performance boost for Snabb. It was made possible by writing customized NIC drivers and DMA access by modern commodity NICs to pass network traffic directly to user space process. The easy to design and implement network function is also another key features of Snabb. The concept of app and linking them together to implement a bigger network function are easy to understand for developers with basic networking knowledge.

Snabb uses Lua scripting language for implementing its back end and front end APIs. Lua being an easy to learn and implement scripting language is another huge advantage for developers to implement their network functionalities. LuaJIT which is a just-in-time compiler for Lua language makes Snabb to optimize the code during runtime. Being a trace based just-in-time compiler, LuaJIT records the execution of program to several traces based on their execution loops and optimizes the traces during runtime. Due to the above reason, LuaJIT is considered to be extremely fast compared to previous Lua interpreter. Snabb is still in constant development cycle and its main author makes feature updates and releases once in a month. Being completely open source, free of license and independent of vendors, its active mailing list attracts network engineers and developers for their contribution.

We also discussed on writing a simple bi-directional forwarding app on Snabb and analysed its throughput and latency characteristics. We saw reduction in throughput and shoot up in latency when number of apps in the forwarding network are increased beyond a threshold. Profiling of forwarding app gave us a good picture of different LuaJIT traces and CPU execution time of those traces. Some of the optimization techniques such as avoiding nested loops, avoiding lots of tiny loops and unpredictable branch conditions could make a significant performance improvement of the program. These optimizations could make the piece of code stay on the same trace and CPU could blindly execute the instructions as long as it stays on single trace.

Lightweight 4over6 'lwaftr' shows one of the practical implementation of Network function virtualization implemented by Snabb. It could be the driving force for most of the network providers to transition their backbone network to IPv6 while still servicing IPv4 customers. Lwaftr is simple and lightweight provider end part of lw4over6 architecture which maps the customers IPv4 address, port ranges and corresponding IPv6 address by using binding table. Lwaftr could handle up to 4 Mpps on line rate over two 10 Gbps Ethernet network interfaces. We also saw that these rates could be increased by increasing the batch pull size of Intel82599 driver app. Making this pull size configurable could give user more control over the traffic handling rate of lwaftr depending upon the amount of network traffic.

Bibliography

[1] Network Function Virtualization (NFV), Architectural Framework.
`http://www-cs-faculty.stanford.edu/~{}uno/abcde.html`

[2] L.Gorrie. Snabb project, Github Snabb code base.
`https://github.com/snabbco/snabb`

[3] Katerina Barone-Adesi, Andy Wingo. [FOSDEM 2016], SnabbSwitch:Riding the HPC wave to simpler, better network appliances.
`https://archive.fosdem.org/2016/schedule/event/snabbswitch/`

[4] Katerina Barone-Adesi, Andy Wingo. [FOSDEM 2016], [*Slides*]. Building fast IPv6 transition mechanisms on Snabb Switch.
`https://wingolog.org/pub/fosdem-2016-lwaftr-slides.pdf`

[5] Deutsche Telekom, IPv6 Transition. Deutsche Telekom tests TeraStream, the network of the future, in Croatia.
`http://www.telekom.com/media/company/168008`

[6] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. Moongen: A scriptable high-speed packet generator. *In Proceedings of the 2015 ACM Conference on Internet Measurement Conference, ACM, 2015.*

[7] MoonGen: A high speed scriptable packet generator, [Github Source].
`https://github.com/emmericp/MoonGen`

[8] Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle. Comparison of Frameworks for High-Performance Packet IO. *In ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2015), May 2015.*

[9] IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. IEEE 1588-2008, July 2008.

[10] L.Gorrie. Snabb Switch: kernel-bypass networking illustrated.
`https://github.com/lukego/blog/issues/13`

[11] L.Gorrie. Snabb Switch's LuaJIT Ethernet Device Driver.
`http://lukego.github.io/blog/2013/01/03/snabb-switchs-luajit-ethernet-device-driver/`

[12] L.Gorrie. Tracing JITs and modern CPUs.
`https://github.com/lukego/blog/issues/5`
`https://github.com/lukego/blog/issues/6`
`https://github.com/lukego/blog/issues/8`

[13] Ntop. PF RING ZC (Zero Copy). [Online], February 2016. Available:
`http://www.ntop.org/products/packet-capture/pf_ring/pf_`
`ring-zc-zero-copy/.`

[14] Mike Pall. The LuaJIT Project, [Running LuaJIT].
`http://luajit.org/`
`http://luajit.org/running.html`

[15] Mike Pall. LuaJIT: How does LuaJIT's trace compiler work.
`http://www.freelists.org/post/luajit/How-does-LuaJITs-trace-compiler-work,`
`1`

[16] Circular Ring Buffer DataStructure.
`https://en.wikipedia.org/wiki/Circular_buffer`

[17] Vhost-User feature for QEMU. *Vhost-User Applied to Snabbswitch Ethernet Switch.*
`http://www.virtualopensystems.com/en/solutions/guides/`
`snabbswitch-qemu/?vos=tech`

[18] Virtio Infrastructure with Vhost-net. *Kernel involvement in Virtio with Vhost-net infrastructure.*
`https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_`
`Linux/7/html/Virtualization_Tuning_and_Optimization_Guide/`
`sect-Virtualization_Tuning_Optimization_Guide-Networking-Virtio_`
`and_vhostnet.html`

[19] Virtual Open Systems. Open Source Virtualization.
`http://www.virtualopensystems.com/`

[20] QEMU Emulation User Documentation. Inter-VM Shared Memory Device
`http://wiki.qemu.org/download/qemu-doc.html#Inter_`
`002dVM-Shared-Memory-device`

[21] Intel DPDK: Data Plane Development Kit.
`http://dpdk.org/`

[22] Universal TUN/TAP device driver.
`https://www.kernel.org/doc/Documentation/networking/tuntap.txt`

[23] Snabb Network Function Virtualization.[Code base]
`https://github.com/snabbco/snabb/tree/master/src/program/snabbnfv`

[24] Snabb Lightweight 4over6. [Code base]
`https://github.com/snabbco/snabb/tree/master/src/program/lwaftr/doc`

[25] Intel Data Direct I/O Technology. [Online]
`http://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html`

[26] L. Rizzo, "Netmap: A Novel Framework for Fast Packet I/O," in 2012 USENIX Annual Technical Conference (USENIX ATC 12). Boston, MA: USENIX, 2012, pp. 101–112.