Diving into Snabb

Dominik Scholz
Betreuer: Paul Emmerich, Daniel Raumer
Seminar Future Internet SS2016
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: scholzd@in.tum.de

ABSTRACT

Virtualization techniques are widely deployed in modern computing applications and in recent years became more interesting for the field of networking. Network Function Virtualization (NFV) is used to replace dedicated hardware solutions for networking appliances like switches, routers and packet filters, with software implementations based on virtualized commodity hardware. Today's network providers plan on replacing their backbone infrastructure with this approach to cope with the increasing demands for bandwidth and mobile communication, the Terastream project of Deutsche Telekom, aiming to deploy a cloud based network with NFV, being a lead example. While this approach results in reduced hardware cost and manifold simplifications achieved through the virtualization, both for developing and deploying services, it poses new challenges to comply with complex constraints of mobile networks, primarily achieving carrier grade performance for network appliances, even when running within virtual machines.

The Snabb virtual switch is a packet processing framework that exploits novel design approaches to fulfil and solve the mentioned requirements and problems, resulting in it being a prime solution for NFV.

Keywords

Snabb, Packet Processing, High-Performance, Virtualization, NFV, Ethernet I/O, Virtual Switch, LuaJIT, SIMD

1. INTRODUCTION

Network Function Virtualization (NFV) [4] has become an increasing trend in recent years, applying the concept of virtualization, the dynamic sharing and aggregating of resources [30], which is already deployed in various computing fields today, to the functionality of networking appliances. Specialized hardware solutions have manifold problems, requiring separate middle boxes for each networking task like routing, packet filtering or DoS protection which not only increases the cost but also the complexity of the system when purchasing technology from multiple vendors. Because of the advances of commodity hardware, allowing to perform networking operations on common x86 hardware with almost equal performance compared to proprietary solutions [14], this can be overcome conveniently by providing virtualized software solutions instead.

The NFV architecture is illustrated in Figure 1. The NFV Infrastructure (NFVI) is based on virtualized commodity hardware, while the different functionalities previously offered through dedicated hardware solutions are instead provided by Virtualized Network Functions (VNF) based on the underlying computing, storage and networking framework. The goals of this approach are to guarantee the same requirements in regard to performance, reliability, stability and security, while improving the manageability. This is achieved by completely separating the software from the underlying hardware and providing dynamic and scalable services which can be flexibly deployed to the network [27]. However, several technical challenges have to be overcome, primarily implementing high performance virtualized networking solutions that are portable to allow integration with hypervisors from different vendors [9]. Furthermore, network services can have complex requirements, for instance in regard to maximum delay.

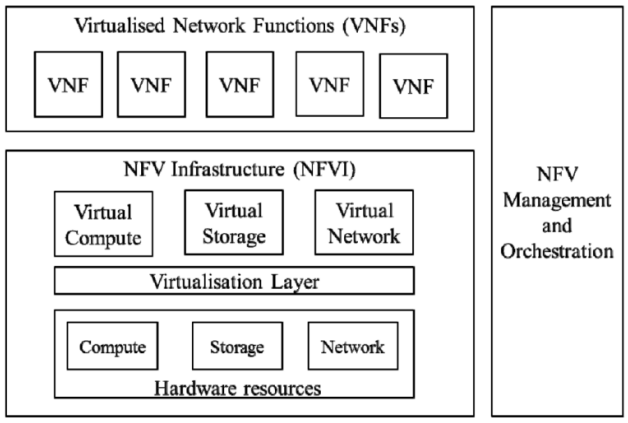

Figure 1: Overview of the Network Function Virtualization Architecture (image taken from [49])

The use cases for NFV technology are manifold, but are particularly interesting for today's telecom core networks [49, 27]. The market situation for network operators has changed in recent years: the demand for mobile communications and Voice over IP (VoIP) has increased, while fixed access lines loose importance [34]. Furthermore, the overall demand for mobile communication and wireless bandwidth is still increasing exponentially, predictions claiming a 10 to 50 fold increase until 2020 [8, 11]. This puts pressure on the network operators to supply scalable and flexible networks which can only be accomplished with a shift to IP-based networks. Deutsche Telekom announced plans to replace their backbone network in the future with a cloud enabled IP network

doi: 10.2313/NET-2016-09-1_05

based on NFV [11] with the Terastream project. By using virtual machines instead of specialized devices and following the "simple - lean - differentiated" [11] approach, the Terastream project aims to not only cope with the increasing demands, but also to reduce complexity and faster deployment of future applications to the customers [11].

Virtual switches are used to virtualize the underlying hardware, for instance, offering virtual Network Interface Cards (NIC) for each network application that make use of one actual NIC [41, 40]. This layer of virtualization has to guarantee the requirements mentioned previously for packet switching and manipulations, primarily carrier grade performance and interoperability for instance with different hypervisors. The Snabb [1] virtual switch [3] is a framework which deploys a novel architecture and exploits virtualization techniques to allow high performance packet processing in user space. Its functionality and goals resulted in a cooperation with the Terastream project to extend Snabb with a dedicated module for NFV, allowing to create VNFs within virtual machines [46]. Therefore, we summarize and explain the core architecture and implemented features of Snabb to understand its compatibility with NFV.

The remainder of this paper is structured as follows. We introduce Snabb, its novel approach in regard to design and architecture, and how to develop and use programs based on this framework in Section 2. The gained insights are compared to other proposals and existing solutions for processing frameworks and NFV in Section 3. In Section 4 the performance of Snabb is demonstrated and briefly analysed by evaluating two programs in a measurement environment. We conclude with a summary in Section 5.

2. SNABB

Snabb [3] is an open source, high-performance networking framework developed by L. Gorrie since 2012. The project's goal is to offer network administrators, Internet service providers and in general operators that have to process large traffic volumes beyond 1GbE software solutions for existing hardware networking appliances [26]. The virtual switch is implemented for modern Linux operating systems, leveraging commodity x86 hardware and Intel NICs [28] running solely in user space [40].

Therefore, Snabb combines new approaches in its core architecture to allow developing of efficient networked applications, which will be illustrated in Section 2.1. The intuitive and easy to use Application Programming Interface (API) as well as code excerpts of sample applications are detailed in Section 2.2.

2.1 Design and Core Architecture

Researchers have demonstrated several bottlenecks of conventional packet processing applications and network stacks in recent years [17, 7, 13], which are in general being avoided and replaced with new techniques by modern high-performance packet processing frameworks (cf. Section 3). Snabb combines these with new approaches to offer fast performing and easy to use network functionalities. The basic paradigm is to focus on simple, small and commodity-targeted solutions instead of complex, large and proprietary ones [5].

The following Sections illustrate the core conceptions that apply these guidelines. The general concept introduced in Section 2.1.1 is also used by other state of the art processing frameworks and not specific to Snabb. A selection of techniques that have not been widely used in networking applications before and are introduced with Snabb are illustrated in Sections 2.1.2-2.1.4.

2.1.1 Kernel Bypass: Ethernet I/O

Various research has shown that using the existing network stack of the kernel, or in general performing networking within the kernel, is not advantageous for a multitude of reasons, including interrupts and context switches for every packet when being passed from the hardware to the kernel and later on to the user space application [17, 7, 13]. Therefore, the trend is to move the complete application to user space, bypassing the kernel. While this achieves the raw performance of the Ethernet device, which can be significantly increased compared to legacy applications [19, 18], it also means that everything that is usually done by the kernel and its inherited features and security measures and guarantees have now to be taken care of in user space.

Kernel bypass networking, which allows to focus on implementing exactly the features beneficial for the application, is a core concept not only chosen by Snabb, but several high-performance packet processing frameworks [17, 16]. The primary idea is to write all low level functionality, the interaction with the hardware, from scratch. Snabb does so for modern Intel NICs. Instead of communicating with the device via the kernel and its drivers, the PCI device is bound directly to Snabb and configured by writing respective values according to the NICs specification using Memory-mapped I/O (MMIO), allowing direct I/O operations. Via Direct Memory Access (DMA) the NIC is allowed to directly access reserved memory regions to read and write packets. In particular, two ring-based FIFO structures for transmission and reception descriptors, storing the addresses of packets to be transmitted or received, respectively, and actual memory for packets are mapped. As the NIC can only operate with physical addresses, *huge pages*, which offer contiguous physical memory blocks of 2 MB or 1 GB, are used for these memory regions [25].

2.1.2 Implemented using LuaJIT

While it is a common approach for packet processing frameworks to focus on the hardware-near C or C++ as primary programming languages and respective established compilers, Snabb uses the lightweight scripting language Lua and the just-in-time LuaJIT [39] compiler instead. Lua is an easy to learn, high-level programming language that has been used in various fields including game development, image processing and robotics [29] for many years. The following illustrates reasons why the use of LuaJIT over another programming language is beneficial for a networking framework.

Firstly, the simple and fast Foreign Function Interface (FFI) allows to easily integrate and work with C libraries [16]. Therefore, Snabb can conveniently make use of low-level

[1] Formerly known as SnabbSwitch

doi: 10.2313/NET-2016-09-1_05

functions and data structures. Secondly, LuaJIT, which produces target specific machine code for instance for x86 and ARM architectures, is a trace based compiler that gathers profiling information during runtime to compile parts of the application in an optimized way [48]. The control-flow is analysed in detail by inspecting whole program paths "to capture the smallest set of execution traces that are representative of the dynamic behavior of the application. Doing so, a trace-based compiler can focus all of its optimization budget on a tiny, yet very important part of an application" [40]. Instead of an one time optimization during traditional static compilation, the just-in-time compiler can adapt the machine code for the current traffic flow or other events, even unexpected ones [40]. These concepts are well researched in the field of compilers [6] and are now applied to networking applications.

Because of these reasons, even low-level code like the driver is written in Lua. Performance evaluation has shown that the results are promising as LuaJIT can outperform applications written in C [35]. The long term goal of Snabb is to drop all references to C functions and structures despite the easy integration with the FFI module and replace them with system-call libraries and assembler libraries written in Lua [5].

2.1.3 Virtualization

Virtualization offers the possibility that real resources can be shared and aggregated amongst multiple virtual applications, optimizing their usage. Furthermore, multiple users using the same physical resource can be isolated to guarantee no interferences and provide security [30]. Applied to virtual machines, networking has to be performed in a fast but also flexible manner, for which only the former applies to hardware NIC virtualization. The Snabb NFV module aims to accomplish both by combining single root IO virtualization (SR-IOV) [32, 12, 36] and a novel virtio technology for the software layer, vhost-user [46].

As Paolino et al. [40] illustrate, Snabb used the pa-ravirtualized I/O framework virtio to connect the kernel-base virtual machines (KVM) with the network controller. This framework is composed of the front end driver virtio-net executed in the kernel of the guest, a backend driver and the emulator module vhost-net, running on the underlying kernel of the host, causing all traffic to go through the virtualized application and the virtual switch. As in this scenario the packets have to pass the kernel of the host, significant overhead is generated by inevitable costly context switches.

vhost-user was implemented to circumvent the host kernel and therefore allow direct communication with the virtual machine. By using Unix domain sockets, with this method in combination with QEMU/KVM, the virtual switch can directly access the shared data structures in the guest's memory space. This allows the transmission and reception of packets without copy operations or context switches, providing carrier grade performance, while also offering flexibility as this is accomplished without specialized drivers. Initially specifically designed for Snabb and its vhost-user App [2] for integration with NFV, Virtual Open Systems has included it in its QEMU release and is now adopted by other virtual switch implementations [40].

The Snabb NFV program exploits these techniques, allowing networking with QEMU/KVM, integrated with or without the OpenStack plugin Neutron at high packet rates [46].

2.1.4 SIMD Offloading

The calculation of, for instance, checksums as they occur in multiple protocols of the ISO/OSI model is a tedious and CPU consuming task, which is why modern NICs allow them to be offloaded to the NIC and calculated in hardware, significantly increasing the performance. Frameworks have started using this approach to improve their performance [16]. However, this comes with the downside that the offloading feature is dependent on the hardware, requiring differing configuration for every NIC, if it even is supported at all.

Snabb uses a different approach as it does not want to be restricted to proprietary hardware features. It makes use of Single Instruction Multiple Data (SIMD) instructions of modern computer architectures. These allow for instruction level parallelism as one instruction leads to the parallel execution of one instruction in multiple functional units. On x86 instruction set architectures the performance of the Advanced Vector Extension (AVX) SIMD instructions has been continuously increased in recent years [23] and is expected to be further improved in the future. Furthermore, SIMD offloading is not depending on the hardware of the NIC, significantly reducing the implementation effort, but also allowing it to be used for other tasks that are usually expensive, like copying of packets [21].

Generalized, the effort of Snabb is to prefer the CPU for tasks over the likes of proprietary NIC features like offloading or Zero-Copy (ZC). With technologies like Intel's Data Direct I/O (DDIO) [1] the data can be directly loaded into the L3 cache of the CPU from which it can easily be accessed and worked with. Even subsequent data copies are cheap, making it possible to use instruction level parallelism over specialized hardware features [5].

2.2 Frontend and Usage

The previous Section illustrated the design approaches of the underlying architecture. Following, the high level architecture and Application Programming Interface (API) which is used by an programmer to develop applications and complete programs are explained.

2.2.1 High-Level API

As already illustrated in the previous Section, Snabb is primarily written in LuaJIT. Scripting languages in general, and Lua in specific, are designed to be easy to learn and use. Using this language not only for the core functionalities but also the end user API allows to write scripts for new programs in an uncomplicated manner. Furthermore, existing features can be understood with less effort and time.

Snabb provides each functionality as a so called "App" in reference to modern App Stores [24]. Their intent is to be

[2] `https://github.com/SnabbCo/snabb/tree/master/src/apps/vhost`

doi: 10.2313/NET-2016-09-1_05

the software counterpart of typical hardware appliances for networking, like an I/O interface, repeater, but also more complex networking devices like switches or routers. Hence, one App represents a specific network function.

Each App has one or more input and outputs links. The typical workflow of an App is to pull packets from an input link, process them according to the scripted task of the App and then push them to an output link. The processing can be anything that translates to actual machine code. These small network functions can then in turn be combined via links, from one App's output to another one's input, to form more complex ones. A link can be compared to an Ethernet cable connecting real network devices. In software it is a ring-based structure used to store the packets between successive network functions [22]. Via this network of separate functions, existing Apps can be reused or, because of the modularized nature, exchanged with new ones [24]. This overseeable and easy to maintain approach allows for the creation of complex network functions based on a network of Apps, all doing a specific task, fitting the NFV approach.

Packets themselves are represented simply by an variable sized array of bytes. This allows to easily work with the data contained as more specific representations, for instance an Ethernet header can be built on top of it [22]. Furthermore, compared to more complex data structures like the *sk_buff* structure used by the Linux network stack, no overhead is created as no additional data, aside from the raw bytes, are stored.

2.2.2 App and App Network Samples

To better illustrate the concepts presented in the previous Section, selected code snippets[3] of the core API functionalities of Snabb Apps and App networks are shown and briefly explained in this Section.

Listing 1 shows the *pull()* function of the Repeater basic App[4]. The purpose of this function is to continuously repeat all packets in the same order received from the input link back to the output link. As described earlier, *push()* is responsible to transmit processed packets back into the network of Apps. In this example, packets are read from the input link and stored in a table (lines 3 sqq.). Then, all gathered packets are replayed back to the output link (lines 7 sqq.).

This general concept of receiving packets, performing arbitrary processing on them and finally transmitting them via outgoing links, like it is with real hardware network devices, is performed by every App of Snabb. Creating an actual program, consisting of a network of these functions, is demonstrated via the *packetblaster* program[5], shown in Listing 2.

The program is used to generate streams of packets, either based on a pcap file or by synthesizing packets with rudimentary configuration options on the fly. The main *run()* function illustrates the general structure of a complete network of functions. The primary task is to generate a configuration for Snabb, containing the definition of the App network, used PCI devices and more. As shown in lines 7 sqq., the program consists of a PcapReader, Repeater and Tee App. The output of the PcapReader is connected with the input of the Repeater (line 10), which in turn is connected to the Tee App (line 11). This App multiplexes the packets to multiple outgoing links, each connected to one of the discovered PCI devices (lines 15 sqq.) as LoadGen App. After the network is configured, the configuration is passed to the Snabb engine and started (lines 25 sqq.), executing the program.

Based on this concept various complex programs have been developed, ranging from the rumpkernel App[6], a complete NetBSD TCP/IP routing stack, to snabbnfv [7], a module for Network Function Virtualization.

```
1  function Repeater:push ()
2     local i, o = self.input.input, self.output.output
3     for _ = 1, link.nreadable(i) do
4        local p = receive(i)
5        table.insert(self.packets, p)
6     end
7     local npackets = #self.packets
8     if npackets > 0 then
9        for i = 1, link.nwritable(o) do
10          assert(self.packets[self.index])
11          transmit(o, packet.clone(self.packets[self.index]))
12          self.index = (self.index % npackets) + 1
13       end
14    end
15 end
```

Listing 1: push() function of the Repeater basic App.

```
1  function run (args)
2     local c = config.new()
3     [...]
4     if mode == 'replay' and #args > 1 then
5        args = lib.dogetopt(args, opt, "hD:", long_opts)
6        local filename = table.remove(args, 1)
7        config.app(c, "pcap", PcapReader, filename)
8        config.app(c, "loop", basic_apps.Repeater)
9        config.app(c, "source", basic_apps.Tee)
10       config.link(c, "pcap.output -> loop.input")
11       config.link(c, "loop.output -> source.input")
12       [...]
13    end
14    [...]
15    pci.scan_devices()
16    for _,device in ipairs(pci.devices) do
17       if is_device_suitable(device, patterns) then
18          nics = nics + 1
19          local name = "nic"..nics
20          config.app(c, name, LoadGen, device.pciaddress)
21          config.link(c, "source."..tostring(nics).."->"
22                         ..name..".input")
23       end
24    end
25    [...]
26    engine.configure(c)
27    if duration then engine.main({duration=duration})
28    else             engine.main() end
29 end
```

Listing 2: Configuration of the App network in the *packetblaster* program

[3] All code snippets are only excerpts from the mentioned sources, shortened and modified for the sake of clarity

[4] https://github.com/SnabbCo/snabb/blob/master/src/apps/basic/basic_apps.lua

[5] https://github.com/SnabbCo/snabb/blob/master/src/program/packetblaster/packetblaster.lua

[6] https://github.com/anttikantee/snabbswitch/tree/rumpkernel

[7] https://github.com/SnabbCo/snabb/tree/master/src/program/snabbnfv

doi: 10.2313/NET-2016-09-1_05

3. RELATED WORK

In addition to the problems induced with virtualizing the hardware, as explained in Section 2.1.3 for the NIC, various performance critical bottlenecks for packet processing frameworks have been assessed. Garcia et al. [17] identified that the allocation of resources on a per packet basis and serialized traffic access are major impediments. Furthermore, the differentiation of user and kernel space implicates multiple copy operations of the packet and costly context switches. They propose improvements to cope with these bottlenecks. Firstly, resources should not be allocated during runtime, but instead preallocated whenever possible. To make best use of the underlying hardware, the whole processing path from receive and transmit queues to applications should be parallelized, whereas each separate path processes a batch of packets at once, reducing per packet overhead. Modern memory mapping techniques in combination with DMA allow for packet processing with zero copy operations.

Multiple frameworks that exploit these features to speed up packet processing have been developed. The Intel Dataplane Development Kit (DPDK) packet processing framework [2] provides drivers and software libraries for Intel based architectures to bypass the data plane of the kernel. It uses optimized data structures implemented in a lockless manner in combination with techniques like memory preallocation and batch processing to speed up processing performance [45]. Furthermore, hardware features of modern Intel NICs are used to offload, for instance, the calculation of checksums to hardware, saving valuable CPU time. One application that is based on the DPDK is the packet generator MoonGen [15]. While focusing on crafting packets and utilizing novel techniques for sub-microsecond timestamping and precise rate control mechanisms, it shows similarities with Snabb in the usage of LuaJIT as front-end scripting language. Packets are generated with Lua scripts, whereby each packet can be separately modified in a flexible way to generate complex streams of traffic. Measurements have shown that MoonGen is capable of saturating a 10 GbE link with minimum sized packets using only one CPU core [16].

The PF_RING ZC (Zero Copy) [38] packet processing framework, developed by ntop, focuses on custom drivers and virtualization. The drivers can be used as replacement for existing kernel drivers, or to bypass the network stack entirely. The Zero Copy technique allows the NIC to directly access memory that the KVM can access, too, removing the overhead normally induced when copying data from the NIC to the kernel and then to user space. This allows for packet processing at 10 GbE even within virtual machines [38].

Specifically aimed at NFV, Martins et al. [33] introduce ClickOS, a platform for virtualized software middleboxes for high performance scenarios with the same goals in regard to flexibility, scalability and performance as NFV. ClickOS utilizes hypervisor virtualization, in particular para-virtualized VMs with the Xen hypervisor, because of its low delay and high throughput. The Click modular router [31] is an architecture intended for creating flexible and configurable software routers at runtime [42] and is used as programming abstraction for ClickOS as it already provides a broad range of networking elements that can be used to create middleboxes, while also being extensible. Instead of running Click in user space or as kernel module, whereas each middlebox is executed within a Linux virtual machine, ClickOS makes adjustments to the Xen hypervisor and implements its own virtual machines based on Click. Briefly explained, each "VM consists of the Click modular router software running on top of MiniOS" [33], whereas MiniOS is a minimized operating system provided by Xen project. Furthermore, utility functions to administer these ClickOS VMs are provided. Extensive and manifold performance analysis of the authors has shown that ClickOS yields the execution of hundreds of middleboxes executing VNFs on commodity hardware, while maintaining low delays. In certain scenarios, a throughput of up to 27.5 Gbps can be reached [33].

Newer versions allow ClickOS to run on top of the packet I/O framework netmap [43], which includes its own virtual switch. Pinczel et al. [42] analysed the performance of a prototype when mapping the proposed architecture of future 5G networks [37] to machines running ClickOS based on netmap. While their model demonstrates good flexibility, Click imposes some limitations in regards to isolation, imposing potential security issues. Furthermore, while the confined programming environment of Click guarantees high performance, integration of third party libraries for the implementation of more complex scenarios is not possible because of restrictions when handling packets, accessing memory or performing I/O operations [42].

4. PERFORMANCE EVALUATION

Snabb provides sample applications itself and even more contributions by the community are accessible in other development branches. In the following Sections two applications were tested in a testbed environment and the results are compared with similar tests of other processing frameworks found in the literature. The measurements were executed in the Baltikum testbed, using a server equipped with a Supermicro X9SCL/X9SCM[8] motherboard, an Intel Xeon E3-1230 v2 CPU clocked at a maximum of 3.3 GHz and an 8 MB L3 cache[9], Dual channel 16 GB ECC DDR3 SDRAM clocked at 1333 MHz and two 82599ES 10-Gigabit SFI/SFP+ NICs[10]. As operating system the Grml Live Linux 2013.02 image with Linux kernel version 3.7 was used. All measurements were performed with the newest release of Snabb, 2016.03 "Tamarillo"[11], unless stated otherwise.

4.1 Traffic Generation: packetblaster

While it is not the primary intent of Snabb, it is able to create and send traffic streams based on pcap files or completely synthesized packets via the *packetblaster* program introduced in Section 2.2.2. As this program is mainly used to test and benchmark other Snabb Apps and programs and therefore should not interfere with such, the packet creation process is different than for other packet generators (cf. Section 3). Instead of allocating and modifying every single packet that is sent, only a few packets are created and written into the transmission queue of the device. When

[8] http://www.supermicro.com/products/motherboard/Xeon/C202C204/X9SCM-F.cfm
[9] http://ark.intel.com/products/65732
[10] http://ark.intel.com/products/32207
[11] https://github.com/SnabbCo/snabb/releases

doi: 10.2313/NET-2016-09-1_05

transmitting a packet, the LoadGen App does not invalidate the buffer, which would free and remove it from the queue, but instead resets the pointer to the DMA memory, basically instructing the NIC to send these packets in an endless loop [20]. This allows the program to generate traffic of multitudes of 100Gbps while only utilizing a small percentage of the CPU. However, this comes at the cost of customizability of the traffic stream. Per default, *packetblaster*, when synthesizing packets, only allows to change Ethernet addresses and the size of IP packets.

The measurement was done with the *packetblaster* program running on the server while the CPU is clocked at 1.6 GHz. The application is instructed to generate as much traffic as possible, crafting minimum sized 60 byte packets[12] using default values and sending them via the two available PCI devices. The output shows that *packetblaster* does so with 10 GbE line rate, reaching the full 14.88Mpps that are theoretically possible for each of the links. Analysing the CPU utilization with the profiling tool *perf* reveals that the program used only one CPU core to do so, however, at a load of 100 %, which would contradict the statement that only a fraction of the CPU is used. This is because the program exploits busy waiting mechanisms[13], exhausting the CPU at all times to avoid the execution of other tasks which would cause costly context switches. Disabling this mechanism and repeating the experiment over a period of 10 seconds with the program pinned to only one CPU core using *taskset* results in approximately 90 million CPU cycles used per second, equalling a utilization of only 5.6%.

For comparison, the dedicated packet generator MoonGen (see Section 3) is able to saturate a 10GbE link using one fully utilized core with customized packets, whereas each successive packet can be uniquely modified [16]. However, it has to be noted that both applications have different purposes: MoonGen aims to generate as much traffic as possible with the given ressources, while Snabb' *packetblaster* is primarily used to test other applications during development and therefore should generate the traffic with the least effort possible.

4.2 Lightweight 4over6: lwAFTR

A more complex program is the implementation of the lightweight 4over6 (lw4o6) architecture [10] lwAFTR. Because the Deutsche Telekom plans to use IPv6-only for their future backbone infrastructure, better tunnelling architectures and protocols that cope with existing problems are necessary to allow efficient encapsulation of IPv4 traffic in IPv6-only networks. This particular solution utilizes 4over6 and NAT. Previously, the primary problem was that keeping the state of the NAT at a centralized point (AFTR) resulted in problems regarding the scalability of the system, because it has to be done per flow. lw4o6 solves this by moving the NAT functionality to the client-side network function (B4) responsible for tunnelling the traffic [10]. The AFTR part only has to maintain a binding of "the [B4's] IPv6 address, the allocated IPv4 address, and the restricted port set" [10]. Therefore, the implementation lwAFTR is a good example to demonstrate the application area of NFV development with Snabb.

Simplified, the *push()* function of the App encapsulates all IPv4 packets coming from the internet and decapsulates all IPv6 packets received from the B4 clients. Neglecting the tasks of handling ICMP packets in a special way or binding lookups in the case of hairpinning [47], the primary operations are memory accesses to copy data when prepending or removing the tunnelling headers, which is a common bottleneck in high-performance packet processing frameworks [17, 45]. The authors claim a performance of 4 Mpps for 550 byte packets resulting in a throughput of 17Gbps on two Intel 82599ES NICs and one core of an Intel XEON CPU clocked at 2.4GHz [5].

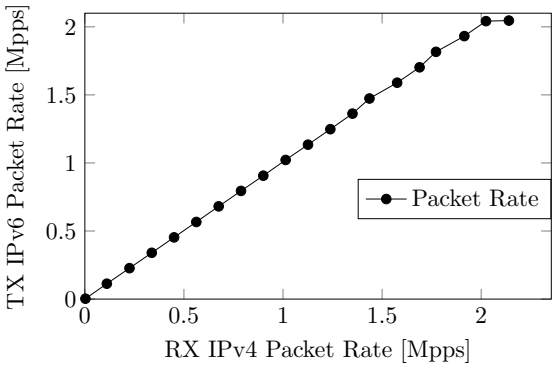

Figure 2: Performance of lwAFTR when encapsulating IPv4 packets

The measurements involved the most recent version of lwAFTR[14] on our test device, clocked at 3.3 GHz. Using another server running the packet generator MoonGen, IPv4 traffic of 550 byte packets matching the binding table with one respective entry was generated and sent to the server running the lwAFTR program. The resulting transmission IPv6 packet rate of the application is illustrated in Figure 2 for different reception IPv4 packet rates. It is clearly visible that the device is able to process all incoming packets until a reception rate of approximately 2.1 Mpps. At this point, the reception rate is approximately 8.4 Gbps with a respective transmission rate of 9.0 Gbps, matching the performance claims presented in [5] mentioned in the previous paragraph.

5. CONCLUSION

In this paper we have introduced the packet processing framework Snabb and its novel design approaches that follow the trends of network function virtualization. It utilizes main-streamed techniques to circumvent common bottlenecks when performing networking operations, for instance bypassing the existing network stack entirely. This can be achieved with DMA and customized NIC drivers. However, instead of using proprietary hardware-based features of modern NICs, like offloading as other frameworks choose to, Snabb uses AVX SIMD instructions to fully utilize the CPU instead, having the advantage of being independent from

[12]The 4 byte *Frame Check Sequence* of the Ethernet frame is appended by the NIC, resulting in a 64 byte frame

[13]See the busywait variable in the file core/app.lua

[14]Commit 2191c16 of `https://github.com/mwiget/snabb/tree/lwaftr_starfruit`

doi: 10.2313/NET-2016-09-1_05

the used NIC. Furthermore, Snabb uses the trace-based LuaJIT just-in-time compiled scripting language based on Lua as primary language, both for back- and front-end programming. Studies have shown that this approach generates optimized code for networking functions on x86 architectures, while the aspect of being an easy to learn and use scripting language eases the development. Because of optimized virtualization techniques, Snabb is eminently for NFV. The novel vhost-user virtio implementation allows packet processing with zero copy operations within virtual machines. These features of the underlying architecture guarantee high-performance processing of packets, while the front-end hides these aspects and provides an easy to use API.

While the project is rather young [15] and the code basis could therefore be considered to still be immature, Snabb enjoys an active community because it is completely independent of vendors, free of license fees and entirely open source. Its activity stretches from mailing lists over websites like reddit to regularly published development blogs, yielding in optimized communication and cooperation amongst projects and developers [26]. The main author publishes a new release every month, steadily extending and improving the core functionality by valuable contributions so that all other developers can incorporate these effortless into their programs [44].

This plays hand-in-hand with the concepts of Apps and programs as a network of Apps. Because of the unified and simple pull and push API of Apps, existing Apps can be reused, while new functionality can be implemented quickly. These small modules, created potentially by many different developers, can then be combined via links to form the complete network. This allows for high use- and reuseability, resulting in a large variety of existing programs.

6. REFERENCES

[1] Intel Data Direct I/O Technology. [Online]. Accessed: March 2016, Available: `http://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html`.

[2] Intel DPDK: Data Plane Development Kit. [Online]. Accessed: March 2016, Available: `http://dpdk.org/`.

[3] snabb-nfv project code base. [Online]. Accessed: March 2016, Available: `https://github.com/SnabbCo/snabb`.

[4] Network Functions Virtualization (NFV); Architectural Framework. [Online], October 2013. Accessed: March 2016, Available: `https://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.01.01_60/gs_NFV002v010101p.pdf`.

[5] K. Barone-Adesi. Snabb Switch: Riding the HPC wave to simpler, better network appliances. Talk at FOSDEM16 [Online], 2016. Accessed: March 2016, Available: `http://mirrors.dotsrc.org/fosdem/2016/ua2114/snabb-switch-riding-the-hpc-wave-to-simpler-better-network-appliances.mp4`.

[6] M. Bebenita, M. Chang, G. Wagner, A. Gal, C. Wimmer, and M. Franz. Trace-based compilation in execution environments without interpreters. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, pages 59–68. ACM, 2010.

[7] R. Bolla and R. Bruschi. Linux software router: data plane optimization and performance evaluation. *Journal of Networks*, 2(3):6–17, 2007.

[8] I. Chih-Lin, S. Han, Z. Xu, Q. Sun, and Z. Pan. 5g: rethink mobile communications for 2020+. *Phil. Trans. R. Soc. A*, 374(2062):20140432, 2016.

[9] C. Cui, H. Deng, D. Telekom, U. Michel, and H. Damker. Network functions virtualisation. *SDN and OpenFlow World Congress*, 2012.

[10] Y. Cui, Q. Sun, M. Boucadair, T. Tsou, Y. Lee, and I. Farrer. Lightweight 4over6: An Extension to the Dual-Stack Lite Architecture. RFC 7596 (Proposed Standard), July 2015.

[11] Deutsche Telekom. Deutsche Telekom tests TeraStream, the network of the future, in Croatia. [Online], December 2012. Accessed: March 2016, Available: `http://www.telekom.com/media/company/168008`.

[12] I. L. A. Division. PCI-SIG SR-IOV Primer - An Introduction to SR-IOV Technology. [Online], January 2011. Accessed: March 2016, Available: `http://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html`.

[13] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 15–28. ACM, 2009.

[14] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, and L. Mathy. Towards high performance virtual routers on commodity hardware. In *Proceedings of the 2008 ACM CoNEXT Conference*, page 20. ACM, 2008.

[15] P. Emmerich. MoonGen. [Online]. Accessed: March 2016, Available: `https://github.com/emmericp/MoonGen`.

[16] 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*, pages 275–287. ACM, 2015.

[17] J. L. Garcia-Dorado, F. Mata, J. Ramos, P. M. S. del Rio, V. Moreno, and J. Aracil. High-Performance Network Traffic Processing Systems Using Commodity Hardware. pages 3–27. Springer Verlag, 2013.

[18] L. Gorrie. Kernel-bypass Networking. [Online], January 2013. Accessed: March 2016, Available: `http://lukego.github.io/blog/2013/01/04/kernel-bypass-networking/`.

[19] L. Gorrie. Snabb Switch's Kernel-bypass Networking. [Online], January 2013. Accessed: March 2016, Available: `http://lukego.github.io/blog/2013/01/05/kernel-bypass-networking-in-snabb-switch/`.

[20] L. Gorrie. [dpdk-dev] TX performance regression caused by the mbuf cachline split. [Online], May 2015. Accessed: March 2016, Available: `http://dpdk.org/ml/archives/dev/2015-May/017506.html`.

[21] L. Gorrie. Packet copies: Expensive or cheap?

[15] Developed since 2012

[Online], October 2015. Accessed: March 2016, Available: https://github.com/SnabbCo/snabb/issues/648.

[22] L. Gorrie. Snabb data structures: packets, links, and apps. [Online], September 2015. Accessed: March 2016, Available: https://github.com/lukego/blog/issues/11.

[23] L. Gorrie. Snabb Switch development: SIMD FTW, or, Straightline redux. [Online], March 2015. Accessed: March 2016, Available: https://groups.google.com/forum/#!msg/snabb-devel/aez4pEnd4ow/WrXi5N7nxfkJ.

[24] L. Gorrie. Snabb Switch in a Nutshell. [Online], August 2015. Accessed: March 2016, Available: https://github.com/lukego/blog/issues/10.

[25] L. Gorrie. Snabb Switch: kernel-bypass networking illustrated. [Online], October 2015. Accessed: March 2016, Available: https://github.com/lukego/blog/issues/13.

[26] L. Gorrie. Why Snabb? [Online], August 2015. Accessed: March 2016, Available: https://github.com/lukego/blog/issues/7.

[27] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee. Network function virtualization: Challenges and opportunities for innovations. *Communications Magazine, IEEE*, 53(2):90–97, 2015.

[28] U. Hoodbhoy. Harnessing the RAW Performance of x86 – Snabb Switch. [Online], October 2014. Accessed: March 2016, Available: http://umairhoodbhoy.net/2014/10/09/harnessing-the-raw-performance-of-x86-snabb-switch/.

[29] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The evolution of lua. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 2–1. ACM, 2007.

[30] R. Jain and S. Paul. Network virtualization and software defined networking for cloud computing: a survey. *Communications Magazine, IEEE*, 51(11):24–31, 2013.

[31] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[32] S. Lowe. What is SR-IOV? [Online], December 2009. Accessed: March 2016, Available: http://blog.scottlowe.org/2009/12/02/what-is-sr-iov/.

[33] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 459–473. USENIX Association, 2014.

[34] T. Mastrangelo. Market Dynamics Forcing Transformation to All-IP Network. [Online], February 2013. Accessed: March 2016, Available: http://blog.advaoptical.com/market-dynamics-forcing-transformation-to-all-ip-network.

[35] C. Mateo. Performance of several languages. [Online], July 2015. Accessed: March 2016, Available: http://blog.carlesmateo.com/2014/10/13/performance-of-several-languages/.

[36] Microsoft. Single Root I/O Virtualization (SR-IOV). [Online]. Accessed: March 2016, Available: https://msdn.microsoft.com/en-us/library/windows/hardware/hh440235(v=vs.85).aspx.

[37] Nokia. Network architecture for the 5g era. [Online]. Accessed: March 2016, Available: http://networks.nokia.com/file/40481/network-architecture-for-the-5g-era.

[38] Ntop. PF_RING ZC (Zero Copy). [Online]. Accessed: March 2016, Available: http://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/.

[39] M. Pall. The LuaJIT Project. [Online]. Accessed: March 2016, Available: http://luajit.org/.

[40] M. Paolino, N. Nikolaev, J. Fanguede, and D. Raho. Snabbswitch user space virtual switch benchmark and performance optimization for nfv. In *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*, pages 86–92. IEEE, 2015.

[41] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker. Extending networking into the virtualization layer. In *Hotnets*, 2009.

[42] B. Pinczel, D. Gehberger, Z. Turanyi, and B. Formanek. Towards high performance packet processing for 5g. In *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*, pages 67–73. IEEE, 2015.

[43] L. Rizzo. Netmap: a novel framework for fast packet i/o. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 101–112, 2012.

[44] M. Rottenkolber. Continuous Integration for Snabb Switch. [Online], November 2015. Accessed: March 2016, Available: http://mr.gy/blog/snabb-ci.html.

[45] D. Scholz, D. Raumer, and F. Wohlfart. A Look at Intel's Dataplane Development Kit. *Proceedings of the Seminars Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)*, NET-2014-03-1 of Network Architectures and Services (NET), August 2014.

[46] snabb.com. Snabb NFV. [Online]. Accessed: March 2016, Available: http://snabb.co/nfv.html.

[47] P. Srisuresh, B. Ford, and D. Kegel. State of Peer-to-Peer (P2P) Communication across Network Address Translators (NATs). RFC 5128 (Informational), Mar. 2008.

[48] C. S. Steele and J. Bonn. Fast functional simulation with a dynamic language. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pages 1–3. IEEE, 2012.

[49] J. Yang. Network Function Virtualization (NFV). [Online], June 2014. Accessed: March 2016, Available: https://jipanyang.wordpress.com/2014/06/16/network-function-virtualization-nfv/.