Comparison of Frameworks for High-Performance Packet IO

Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart,
Daniel Raumer, and Georg Carle
Technische Universität München, Department of Informatics
Chair for Network Architectures and Services
Boltzmannstr. 3
85748 Garching b. München, Germany
{gallenmu | emmericp | wohlfart | raumer | carle}@net.in.tum.de

ABSTRACT

Network stacks currently implemented in operating systems can no longer cope with the packet rates offered by 10 Gbit Ethernet. Thus, frameworks were developed claiming to offer a faster alternative for this demand. These frameworks enable arbitrary packet processing systems to be built from commodity hardware handling a traffic rate of several 10 Gbit interfaces, entering a domain previously only available to custom-built hardware.

In this paper, we survey various frameworks for high-performance packet IO. We introduce a model to estimate and assess the performance of these packet processing frameworks. Moreover, we analyze the performance of the most prominent frameworks based on representative measurements in packet forwarding scenarios. Therefore, we quantify the effects of caching and look at the tradeoff between throughput and latency.

Categories and Subject Descriptors

C.2 [**Computer-Communication Networks**]: Miscellaneous; C.4 [**Performance of Systems**]: Measurement techniques

General Terms

Measurement, Performance

Keywords

netmap; PF_RING ZC; DPDK; software packet processing; performance measurement

1. INTRODUCTION

Nowadays, 10 Gbit Ethernet adapters are commonly used in servers. However, due to overhead imposed by the network stacks' architectural design the CPU quickly becomes the bottleneck, so that packet handling – even without any complex processing – is impossible at line speed for small packet sizes. Software frameworks for high-speed packet IO, e.g. netmap [1], Intel DPDK [2], or PF_RING ZC [3], promise to fix this issue by offering a stripped-down alternative to the Linux network stack. Their performance increase allows using commodity hardware systems as routers and (virtual) switches [4, 5], network middleboxes like firewalls [6], or network monitoring systems [7]. Motivated by the potential gain we analyzed the performance characteristics of these frameworks.

Initially, we investigate the factors network bandwidth, CPU performance, PCI express bandwidth, and connection to main memory influencing the performance of arbitrary packet processing applications. These factors are comprised into a model describing this type of program based on the previously mentioned frameworks. Various measurements show the applicability of this model, with packet forwarding as the basic test scenario. Each measurement is designed to investigate the influence of a specific factor on the forwarding throughput, e.g. the clock speed of the CPU, the number of processed packets per call or the cache utilization. Moreover, the latency of packet forwarding is also reviewed. All measurements are designed to ensure the comparability of our test results: running on the same CPU, equipped with the same 10 Gbit NICs while using packet forwarders applying the same algorithm for each of three frameworks respectively to provide a fair comparison between them.

The paper is organized as follows: In Section 2, we describe the state of the art in fast packet processing. Related work that serves as basis for our research is presented in Section 3. Section 4 identifies bottlenecks for packet processing in software and derives a model describing such applications. Subsequently, Section 5 presents our comparison of techniques for fast packet processing. We conclude with a summary of our results in Section 6.

This paper is based on a master's thesis [8] offering a more comprehensive discussion including the usability of the frameworks and a closer description of the APIs.

2. PACKET PROCESSING IN SOFTWARE

Network traffic processing performance backed by commodity hardware systems has increased continuously in the last years. The increase came both from software optimizations and hardware developments like the move from 1 Gbit to 10 Gbit Ethernet, multicore CPUs and offloading features that save CPU cycles.

2.1 Utilization of Hardware Features

On the hardware side the performance increase, beside the higher bandwidth, came from offloading features that allow shifting workload from the CPU to the Network Interface Card (NIC). Checksum offloading, for instance, relieves the CPU from CRC checksum handling. Now the NIC takes care of calculating the CRC checksum and adding it to the packet before transfer. On the receiving end, the NIC validates the checksum and drops the packet in case of an error, without involving the CPU. [9]

Direct Memory Access (DMA) allows the NIC to write or read packets directly into or from RAM bypassing the CPU. Modern NICs can even copy packets directly into the cache of the CPU, which leads to a further increase in performance [10].

Another part of the speed-up comes from increased CPU performance. In addition to higher clock rates, the number of CPU cores has changed from one to a growing number of cores. NICs need to support multi-core architectures explicitly by distributing incoming packets of different traffic flows to different cores. One of these techniques is Receive Side Scaling (RSS), which allows scaling packet processing with the number of cores. [9]

2.2 Linux Network Stack

On the software side the performance increase came, despite the support of the new hardware features, from a more efficient way to handle incoming traffic. The first approach of generating one interrupt per incoming packet was unsuitable for high packet rates due to livelocks caused by interrupt storms [11]. In such a livelock the system is almost fully occupied with handling the overhead caused by interrupts instead of processing packets.

NAPI, a network driver API introduced in kernel 2.4.20, reduces the number of interrupts generated by incoming traffic with the ability to switch to polling for packets during phases of high load, effectively reducing system overhead [11]. The NAPI-based network stack is sufficiently powerful to scale software routers to multiple Gbit/s [12,13]. But even though performance improved, the Linux network stack primarily focuses on offering a fully featured general-purpose network stack for an operating system (OS) rather than providing an interface for optimal performance needed for software router applications [14].

2.3 High-Speed Packet Processing

Compared with a general-purpose network stack like implemented in Linux, high-speed packet IO frameworks offer only basic IO functionality: Layer 3 and above must be implemented by the application whereas the Linux network stack handles layer 3 and 4 protocols like IP and TCP. As a benefit, these frameworks offer increased performance compared to a full-blown network stack. In this paper we focus on the most important representatives netmap [1], PF_RING ZC [15], and Intel DPDK [2]. All three frameworks require modified drivers and use the same techniques for acceleration:

- Bypassing the default network stack, i.e. the packets are only processed by the processing framework and by the applications running on top of them.
- Relying on polling to receive packets instead of interrupts.
- Preallocating packet buffers at the start of an application with no further allocation or deallocation of memory during execution of an application.
- No copying of data between user and kernel memory space as a packet is copied once to memory via DMA by the NIC and this memory location is used by processing framework and applications alike.
- Processing batches of packets with one API call on reception and sending.

netmap.

netmap [1] exposes packet buffers to the application and uses standard system calls, like `poll()` or `ioctl()`, to initiate the data transfer. The work behind these system calls is reduced compared to a default network stack. These system calls only update the packet buffers and check the data provided by user programs for their validity to prevent crashes.

The network drivers of netmap are based on regular Linux drivers. As long as no netmap application is active the driver works transparently for OS and traditional applications. Upon starting a netmap-enabled application the NIC is put into in a special "netmap mode", i.e. the NIC becomes inactive for the OS and no packets are delivered to the standard OS interfaces and traditional applications. Instead, the packets are transferred to netmap specific data structures where they are available to the netmap-enabled application. When closing this application the driver switches back to transparent mode. Maintaining this compatibility in the driver allows for an easy integration into general-purpose operating systems. netmap has already been integrated into the FreeBSD kernel [16] and inclusion in the Linux kernel has been discussed [17].

Multiple applications have shown increased performance by adapting netmap: Click [18] a software router, the virtual switch VALE [4], and the FreeBSD firewall ipfw [6].

A notable difference of the different network APIs is the usage of system calls. Linux does the entire packet handling in kernel space to ensure a high degree of security and robustness. DPDK and PF_RING ZC perform their packet processing entirely in user space to provide high performance. To provide robust and fast packet processing netmap combines both approaches. Most of the workload, i.e. packet processing, is done in user space. System calls perform only basic checks on the packet buffers to initiate reception and transfer of packets.

PF_RING ZC.

PF_RING ZC does not use standard system calls but offers its own functions. The API of PF_RING ZC emphasizes a convenient multicore support [19]. It is NUMA aware, i.e. on systems with multiple CPU sockets the packet buffers can be allocated in memory regions a CPU can directly access. Moreover processes can be clustered for easy data sharing amongst them.

PF_RING ZC features a driver with capabilities similar to those of netmap, i.e. the driver is based on a regular Linux driver acting transparently as long as no special application is started. During the time such an application is active regular applications cannot send or receive packets using the respective default OS interfaces. This driver may also be configured to the deliver a copy of the packets to be

available to the OS, while a PF_RING ZC application is active, but the duplication process lowers the performance.

Ntop [20] offers a number of applications running on top of PF_RING ZC. For example n2disk, a packet capturing tool, or nProbe, a tool for traffic monitoring.

DPDK.

DPDK is a collection of libraries, which offers not only basic functions for sending and receiving packets, but also provides additional functionality like a longest prefix matching algorithm for the implementation of routing tables and efficient hash maps. It relies on a custom user space API similar to PF_RING ZC instead of traditional system calls used by netmap. The DPDK API [21] offers multicore support, additional libraries used for packet processing, and features the highest degree of configurability amongst the investigated frameworks.

The driver of DPDK does not feature a transparent mode, i.e. as soon as this driver is loaded, the NIC becomes available to DPDK but is made unavailable to the Linux kernel regardless of whether any DPDK-enabled application is running or not. DPDK uses a special kind of driver aiming to do most of its processing in user space. This UIO driver [22] still has a part of its code realized as kernel module but its tasks are reduced. It only initializes the used PCI devices by mapping their memory regions into the user space process.

A notable example for an application using DPDK, which gained attention, is an accelerated version of Open vSwitch called DPDK vSwitch [5]. An additional high-performance software switching solution is xDPd [23], which supports DPDK for network access.

Other Frameworks.

The already mentioned frameworks are not the sole solutions offering high-speed packet processing capabilities in user space.

PacketShader [24] is a packet processing framework using the general purpose capabilities of GPUs for packet processing. It also features a separate engine for fast packet IO. The GPU part of PacketShader is not publicly available, only the code of the packet engine was released, which can be used on its own. However, this engine is not developed as actively as netmap, PF_RING ZC, or DPDK leading to a low number of updates in the repository [25].

PFQ [26] is a framework that is optimized for fast packet capturing. It does not rely on specialized drivers like the other frameworks and can be used with every NIC as long as this card is supported by Linux. However, without modified drivers the NIC cannot push the packets directly to user space. The lack of this feature leads to a performance disadvantage when compared to the other frameworks presented by our paper. A notable feature of PFQ is the integration of a Haskell-based domain specific language for implementing packet processing algorithms [27]. PFQ focuses on providing a framework to make packet processing easy and safe rather than providing the highest possible performance. Therefore the typical use cases for PFQ differ from the use cases of the other frameworks.

Snabb Switch [28] aims to combine powerful packet processing capabilities with the scripting language Lua. Using a scripting language lowers the entry barriers for developers, as these languages are designed to be learned easily and also allow developing applications with few lines in little time. Snabb Switch's design philosophy even applies to the driver as it is also written in Lua. In 2012 this framework was released. Being the youngest framework in this enumeration it is not as mature as the other frameworks, also no applications are known to use Snabb Switch as their backend.

The three frameworks presented in this paragraph were excluded from a thorough examination for various reasons. For PacketShader the reasons were the unavailability of the GPU part and low development activity. PFQ was not investigated because of different design goals leading to a performance disadvantage compared to the other frameworks. Snabb Switch was excluded because of its immaturity. This paper only focuses on the three most established competitors: netmap, PF_RING ZC, and DPDK, which appear to be more mature, leading to a higher availability of applications using the three presented frameworks.

3. RELATED WORK

A survey of various packet IO frameworks was published by García-Dorado et al. [14]. The theoretical part of their investigation is quite comprehensive and the paper includes measurements showing selected aspects of these frameworks, e. g. the influence of the number of available cores and packet sizes on the throughput. They investigate the packet IO engine of PacketShader, PFQ, netmap, and PF_RING DNA, a predecessor of PF_RING ZC. DPDK and Snabb Switch are not investigated. However, the authors only analyze packet capturing capabilities and neglect other aspects of packet processing.

Throughput measurements of software packet forwarding systems on commodity hardware have been conducted previously: Bolla and Bruschi analyze a Linux software router [13]. Studies of software router performance and the influence of various workloads were published by Dobrescu et al. [12]. The highest throughput of a software solution implementing an OpenFlow switch with DPDK was presented and measured in [29]. We also measured the throughput of Linux-based forwarding tools in previous work [30]. These measurements allow for a direct comparison with results from this paper because they were performed on the same test system.

The latency of a Linux software router was also measured by Bolla and Bruschi in [13]. A technique to measure different parts of packet processing systems using commodity hardware based on an understanding internal queuing was described by Tedesco et al. [31]. Rotsos et al. [32] present a FPGA-based method to measure the latency of various software and hardware OpenFlow switches. They present measurements for Open vSwitch running on Linux as an example. Discussion of latency in software routers can also be found in [33]. The authors describe a method that can be used to distinguish the latency introduced by queuing from the processing delay.

The selected literature shows that the performance of the Linux networking part is thoroughly researched and well-known. There are also papers investigating a specific framework exclusively. However, measurements require similar test conditions, i.e. comparable hardware and software setups, to ensure comparability. The paper by García-Dorado et al. [14] provides those conditions but measures only a few selected aspects. Therefore we try to give a fair comparison by testing each framework on the same hardware. This

paper includes measurements not yet published in similar papers, e.g. the transmission of packets or latency determination. We also introduce a new model to provide a basic understanding how packet processing applications work and how their performance can be estimated.

4. PERFORMANCE CONSIDERATIONS

We present a model that provides insights into the performance of the packet processing applications built for high-speed IO frameworks. It uses the main factors influencing performance to provide an upper bound for the capabilities of a software-based packet processing system and to show the limits and potential bottlenecks.

4.1 Limits and Influencing Factors

Performance limits are grounded on four different characteristics of the hardware:

1. The maximum transfer rate of the used NICs. Thus it is determined by the Ethernet standards (i.e. 1 GbE, 10 GbE, or 40 GbE).

2. PCI express is used to connect the NICs to the rest of the system. Nowadays typical hardware uses PCIe v2.0 with 8 lanes per NIC, which offers a usable link bandwidth of 32 Gbit/s [34] for every interface in rx and tx direction respectively. Commonly available two port NICs cannot reach the 32 Gbit/s limit, which renders this limit irrelevant for this type of NIC.

3. As packet data is sent to the memory the RAM could restrict the possible network bandwidth. A typical system with DDR3 memory provides a bandwidth of 21.2 Gbytes/s (dual channel, effective clock speed of 1333 MHz [35]. Our measurements showed that this bandwidth is high enough to support at least eight network ports transferring and sending concurrently at 10 Gbit/s. In more sophisticated hardware setups, i.e. servers with several CPUs, the actual configuration may limit the achievable transfer rates. If a CPU has to access a NIC or RAM attached to a different CPU, the interconnect between CPUs may act as a bottleneck.

4. The fourth component involved in packet processing is the CPU. Due to modern offloading features of NICs the processing load on the CPU can be kept low. Examples like netmap show that handling a fully loaded 10 Gbit/s link is possible even if the traffic consists of a high number of short packets. This is not the case if the Linux network stack is used. However, if complex packet processing algorithms are performed, the CPU may lower the transfer rate even for high-speed packet frameworks. Therefore the CPU is considered to be the dominating bottleneck. [1]

4.2 Upper Bound for Packet Processing

In this section, we use the identified bottlenecks to conduct a generic model for packet processing. These limits act as upper bounds for the number of packets per second that can be processed.

The first upper bound is a fixed limit determined by the used Ethernet standard (cf. Point 1 of enumeration in Section 4.1). This limit is called r_{max}.

The second upper bound depends on the CPU of the packet processing system (cf. Point 4 of enumeration in Section 4.1). That computational limit is called c_{max}.

For simplicity we only consider these two bounds and omit the upper bound for memory and PCIe bandwidth, as these are not a bottleneck at least in our test setup. A combination of the upper bounds leads to the maximum number of packets per second that can be processed, also known as upper bound of the throughput T_{max}. As the upper bound of the throughput cannot exceed a single upper bound, T_{max} can be described by the following formula:

$$T_{max} = MIN(r_{max}, c_{max}) \tag{1}$$

The value t_{max} itself is influenced by two factors, the line-rate allowed by the Ethernet standard $v^{ethernet}$ and the individual packet size s_i^{packet} (this size also includes bits for preamble, start frame delimiter, padding and inter frame gap).

$$v^{ethernet} \geq \sum_{i=0}^{n} s_i^{packet} \tag{2}$$

If n in Equation 2 is maximal, it equals to r_{max}, i.e. n is the number of packets that can be sent per second with respect to their individual packet length s_i^{packet} and the bandwidth of the Ethernet $v^{ethernet}$.

The value for c_{max} depends on the resources provided by the CPU - the cycles. These processing cycles can either be used to handle packets or to process other tasks. The following formula uses f^{CPU} to describe the available number of cycles provided by the CPU per second. Costs for an individual packet are represented by c_i^{packet}. All costs for other processing tasks running on the CPU are summed up in c^{other}.

$$f^{CPU} \geq c^{other} + \sum_{i=0}^{n} c_i^{packet} \tag{3}$$

The total costs of the packet processing in CPU cycles is given by the sum in Equation 3. This sum contains the number of processed packets represented by the number of packets per second n and the individual costs c_i^{packet} of the packets. The maximum number of cycles per second f^{CPU} is a fixed value depending on the hardware. Therefore, the CPU resources available for packet processing and the other system tasks are bound by this limit, i.e. they have to be smaller or equal than f^{CPU}. To get the maximum number of packets c_{max}, n has to be maximized with respect to Inequation 3.

Figure 1 shows the combination of the two upper bounds T_{max} as combination of r_{max} and c_{max} in respect to growing costs per packet described by the x-axis. For this section only the dashed and dotted lines are relevant. The value r_{max} depends on the size of the packets. When using minimally sized packets with 64 Byte and taking overhead data like preamble or inter frame gap into account, 14.88 million packets per second (Mpps) can be achieved on 10 GbE. As long as r_{max} is reached, the costs c_{max} are low enough to be fully handled by the available CPU, the traffic is bound by the limit of the NIC. At the point c_{equal} the throughput begins to decline. Beyond this point, processing time of

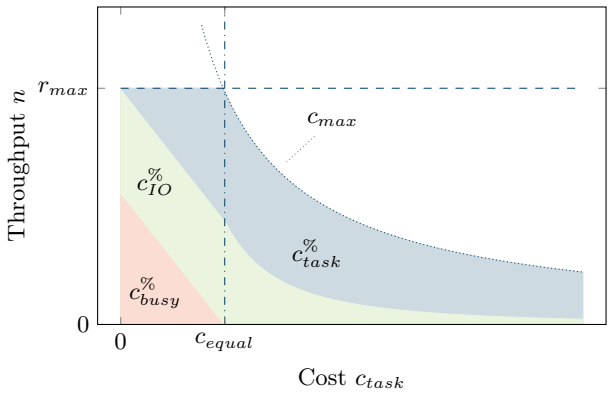


Figure 1: Model for packet processing

the CPU does not suffice the traffic capabilities of the NIC, i.e. the traffic becomes CPU bound and the throughput subsequently sinks.

The costs per packet determine how many packets can be processed without surpassing the computational limit c_{max}. The actual shape of c_{max} cannot be determined as it depends on the traffic and the processing task. Regardless of the precise shape of this curve, the outcome stays the same, i.e. higher per-packet costs decrease the throughput. The hyperbolic shape of c_{max} depicted in Figure 1 holds for packet processing frameworks and is explained in detail in the following section.

4.3 High-Performance Prediction Model

According to Rizzo, packet processing costs can be divided into per-byte and per-packet costs, with the latter dominating for IO frameworks; i.e. it is only slightly more expensive to send a 1.5 KB packet than sending a 64 B packet [1]. This leads to two assumptions to be made. The first assumption is that the per-packet costs are constant for high-performance IO frameworks. The second one is that experiments are performed under the most demanding circumstances if the highest packet rate is chosen, i.e. 64 B packets have to be used.

In case of constant costs per packet $\forall i \; : \; c_i^{packet} = c_{const}^{packet}$ and a dedicated core for packet processing leads to $c^{other} = 0$. If the packet processing application itself also generates a constant load per packet and the high-performance frameworks have roughly constant costs per packet and use dedicated cores for packet processing the Inequation 3 can be simplified to the following inequation:

$$f^{CPU} \geq n \cdot c_{const}^{packet} \tag{4}$$

If packet processing includes actions that depend on the type of packet or the traffic characteristics, computation may become infeasible. Such a scenario may be packet monitoring were certain types of packets require additional CPU cycles for further analysis [36]. Without restriction to certain traffic patterns it is still possible to approximate the overall costs with average per-packet costs or to do a worst case estimation.

Due to the architecture of the frameworks, which all poll the NIC in a busy waiting manner, an application uses all the available CPU cycles all the time. If the limit of the NIC is reached but $n \cdot c_{const}^{packet}$ is lower than the available CPU cycles, the cycles are spent waiting for new packets in the busy wait loop. If these costs are included, a new value is introduced $c_{const}^{*packet}$ and both sides of the former Inequation 4 are now balanced:

$$f^{CPU} = n \cdot c_{const}^{*packet} \tag{5}$$

The costs per packet $c_{const}^{*packet}$ can originate from different sources:

$$c_{const}^{*packet} = c_{IO} + c_{task} + c_{busy} \tag{6}$$

1. c_{IO}: These costs are used by the framework for sending and receiving a packet. The framework determines the amount of these costs. In addition, these costs are constant per packet due to the design of the frameworks by completely avoiding operations depending on the length of the packet, e.g. buffer allocation.

2. c_{task}: The application running on top of the framework determines those costs, which depend on the complexity of the processing task.

3. c_{busy}: These costs are introduced by the busy wait on sending or receiving packets. If throughput is lower than t_{max}, i.e. the per-packet costs are higher than c_{equal}, c_{busy} becomes 0. The cycles spent on c_{busy} are effectively wasted as no actual processing is done.

Combining Equations 5 and 6 leads to:

$$f^{CPU} = n \cdot (c_{IO} + c_{task} + c_{busy}) \tag{7}$$

Figure 1 depicts the behavior of the throughput while gradually increasing c_{task} as described by Equation 7. The highlighted areas show the relative part of the three components of $c_{const}^{*packet}$. Each area depicts the accumulated per-packet costs of their respective component x called $c_x^{\%}$.

The relative importance of $c_{IO}^{\%}$ compared to $c_{task}^{\%}$ decreases for higher task difficulties because of two reasons. The first reason is the decreasing throughput with fewer packets needing a lower amount of processing power. The second reason is that while c_{task} increases the relative portion of cycles needed for IO gets smaller.

Low values of c_{task} and only parts of the cycles spent on c_{IO}, increase busy waiting that leads to a high value for c_{busy}. $c_{busy}^{\%}$ decreases linearly while $c_{task}^{\%}$ grows accordingly until c_{equal} is reached. This point subsequently marks the cost value, where no cycles are wasted on busy waiting.

c_{task} increases steadily, which leads to a growing relative portion of $c_{task}^{\%}$.

5. PERFORMANCE COMPARISON

The available CPU cycles are the main limiting factor of software packet processing (cf. Section 4.1). Subsequently, the throughput of a packet processing application heavily depends on the amount of CPU cycles available for its processing task. This amount is influenced by numerous factors and the following measurements present a selection of factors we consider relevant for real world applications: The overhead caused by the complexity of packet processing, the time the CPU spends waiting for data to arrive in cache, and the effect of different batch sizes, i.e. whether the packet throughput rises if more packets are processed per call. For

every factor a dedicated measurement is performed. As batch size in particular determines the queuing delay of the packets on the processing system and latency during packet forwarding is also investigated.

Initially, we explain the test setup and various methods to precisely determine the used CPU cycles and check them for the applicability for our tests.

5.1 Measurement Setup

Our test setup consists of three distinct servers: a forwarder running the investigated frameworks, a source, and a sink connected via 10 GbE links. The device under test is equipped with a dual port Intel X520-SR2 NIC, the load generator and sink use single port X520-SR1 NICs. These cards use PCIe v2.0 with 8 lanes, which offers a usable link bandwidth of 32 Gbit/s in both directions. The Intel cards were chosen, as driver implementations exist for each of the investigated frameworks. Also possible performance influences introduced by different NICs are avoided. The server acting as forwarder runs on an Intel Xeon E3-1230 V2 CPU. The clock speed was fixed at 3.3 GHz, with power conserving mechanisms, Turbo Boost, and Hyper-Threading deactivated to make the measurements consistent and repeatable.

The forwarder statically forwards packets between the two interfaces without consulting a routing or flow table. It modifies a single byte in the packet to ensure that the packet is loaded into the first level cache. Forwarding is done in a single thread pinned to a specific core.

As performance depends on the number of processed packets rather than the length of the individual packets, we use constant bit rate traffic with the minimum packet size of 64 B for all measurements in this paper to maximize the load on the frameworks. The packets are counted on the sink using the statistics registers of the NIC.

We conducted measurements with a version of netmap, which was published on the 23rd March 2014 in the official repository [37], PF_RING ZC version 6.0.2 [20] , and DPDK version 1.6.0 [38].

Our packet generator MoonGen [39] was used for latency measurements. It uses hardware features of our Intel NICs for sub-microsecond latency determination.

Every data point in our performance measurements is an average value. This value is calculated from 30 single measurements over a period of 30 s. Confidence intervals are unnecessary as results are stable and reproducible for all frameworks. An observation also made by Rizzo in the initial presentation of netmap [1].

5.2 Determine the Transmission Efficiency

In our testbed all of the frameworks are able to forward packets at full line rate with a single CPU core. To measure the transmission efficiency expressed by the CPU load caused by packet transmission, the CPU load generated by each framework needs to be compared. In Equation 7 this efficiency is referred to as c_{IO}. A low number of cycles spent on c_{IO} increases the number of cycles available for the actual packet processing task, i.e. c_{task}. This in return allows more demanding applications to be built using more efficient frameworks without performance penalties.

5.2.1 Known Approaches for Measuring CPU Load

However, due to their architecture (cf. Section 2.3), excessive polling on the NIC causes the CPU cores used by the frameworks to be under full load at all times. Therefore, a simple comparison of CPU usage by a tool like `top` does not work. For this kind of measurement there is no way to tell the relative portions of the three components of $c_{const}^{*packet}$ in Equation 7 apart.

The use of a profiling tool would list the relative portion of each called function. By adding up the result for the functions associated with efficiency c_{IO}, this component could be determined. This method was also rejected as the overhead introduced by the interruption caused by the profiling tool itself lowers the throughput and affects the measurement.

Rizzo measured efficiency by reducing the CPU clock frequency until the throughput of the NIC was beginning to decline [1]. At this point no busy waiting cycles happen as depicted in Figure 1. This results in a c_{busy} value of 0. The packet processing task was simplified so that this component named c_{task} can also be neglected. Only component c_{IO} remains, which is the efficiency of the framework. However even at the lowest supported clock speed (1.6 GHz) in our test setup the forwarders transmitted at full line rate. Therefore this solution could also not be applied.

5.2.2 Novel Method

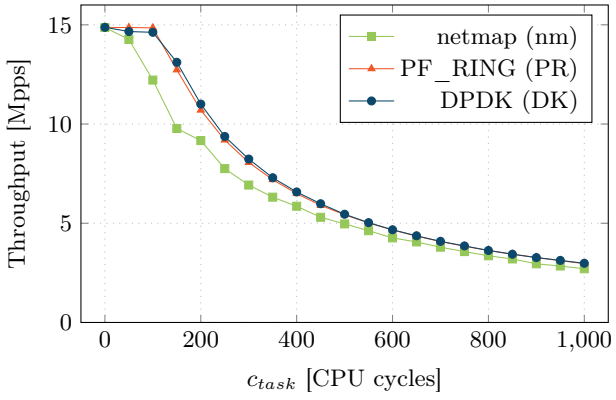
To overcome the flaws of the previously presented methods for determining the efficiency, we introduce a novel method for our measurements. Therefore we add a piece of software producing a constant load per packet on the CPU. The load can be specified as a number of CPU cycles to wait. This value can be increased until the throughput begins to decline. Intel provides a benchmark method [40] based on a clock counter called TSC. We used this guide to design and calibrate this load mechanism. The code containing the load generation and benchmarking mechanisms is available at [41].

At the point of decline c_{busy} is known to be 0, c_{task} is known by design. Subsequently c_{IO} can be calculated. For this experiment the forwarders were modified to implement this emulated CPU load c_{task} by spending a predefined number of CPU cycles per packet beside the framework's packet IO operations. For the basic performance tests we ignore cache effects that can occur during a lookup in a data structure (e.g. a forwarding or flow table). Therefore, we can assume that packet processing applications spend a fixed amount of CPU cycles per packet.

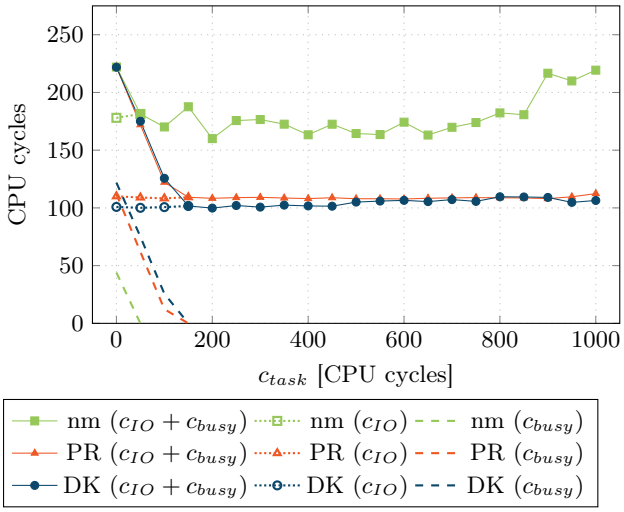
5.3 Measure the Transmission Efficiency

Figure 2(a) presents the results of throughput measurements with different CPU loads for the task emulator. As anticipated by our model in Figure 1, an increasing workload decreases the measured throughput. In the next step, we get back to our goal of measuring the per-packet CPU load consumed by each framework. To forward a packet, a CPU core must dedicate a number of cycles for transmission (c_{IO}), i.e. for receiving and sending a packet, cycles for the emulated task (c_{task}) and possibly polls the NIC unnecessarily (c_{busy}).

Knowing f^{CPU}, and taking r and c_{task} from Fig. 2(a) allows for the calculation of $c_{busy} + c_{IO}$ by applying our Equation 7. These results are shown in Figure 2(b) for each framework. Starting at around 220 cycles for $c_{busy}+c_{IO}$ the graph decreases until the throughput is not longer limited by the 10 Gbit/s line rate. At this point, the throughput becomes limited by the CPU and no busy wait cycles happen



(a) Forwarding with an emulated task



(b) Transmission cycles c_{IO} & busy polling cycles c_{busy}

Figure 2: Transmission efficiency measurements

any longer, i.e. $c_{busy} = 0$. This allows for the separation of the two components c_{busy} and c_{IO} into two individual graphs also depicted in Figure 2(b).

netmap becomes CPU bound with 50 cycles of additional workload per packet, DPDK and PF_RING ZC after 150 cycles. At this point c_{IO}, which describes the cycles needed for a packet to be received and sent by the respective framework, reaches its lowest value and stays roughly constant for all higher packet rates. DPDK has the lowest CPU cost per packet forwarding operation with approximately 100 cycles.

We measured a c_{IO} of approximately 900 cycles for forwarding applications based on the Linux network stack in previous work [30]. This means that the frameworks discussed in this paper can lead to a nine-fold performance increase over classical applications.

5.4 Influence of Caches

The forwarding scenarios in the previous section ignored the influence of caches, which can introduce a delay when accessing a data structure, e.g. the routing table. To imitate this behavior the task emulator described in the preceding section was enhanced to access a data structure while transferring packets.

The necessary time to access data residing in RAM is shortened by the ability of modern CPUs to buffer accesses to RAM by integrating a hierarchy of several caches differing in size and access time. To test for different scenarios with only partly filled caches the size of the data structure was made adaptable. The software influences what is put into cache indirectly by accessing data in RAM, which is then put into the cache or by giving hints to memory addresses via specialized commands. To optimize for common access patterns, data close to already accessed addresses can be prefetched by the CPU before it is accessed [42]. Our tests showed that if a data structure is accessed linearly this prefetching is working efficiently enough to hide the slow access speed to RAM. In the scenario of a routing table the data to be accessed is determined by the traffic and the access pattern is likely to be non linear.

To mimic a worst-case scenario the addresses accessed were randomized. Aiming for a realistic scenario the prefetching was counteracted by using a circular linked list with random access pattern. This was achieved by randomly chosen links between the list elements while ensuring that the permutation contains a single cycle so that all memory locations are accessed when the list is traversed. This guarantees a random access on RAM or cache by iterating one step through the list for each received packet. The size of the linked list can be varied to emulate different routing or flow table sizes. An implementation of this data structure is publicly available [41].

Figure 3(a) depicts the throughput of the investigated frameworks in relation to the list size of our task simulator. For every packet processed one emulated table lookup was performed. To investigate CPU-limited, rather than NIC-limited, throughput a constant CPU load of 100 cycles was introduced, the point in Figure 2(a) where the throughput was beginning to decline for all three frameworks. This offset explains the lower throughput of netmap in Figure 3(a), as expected from the data in Figure 2(a).

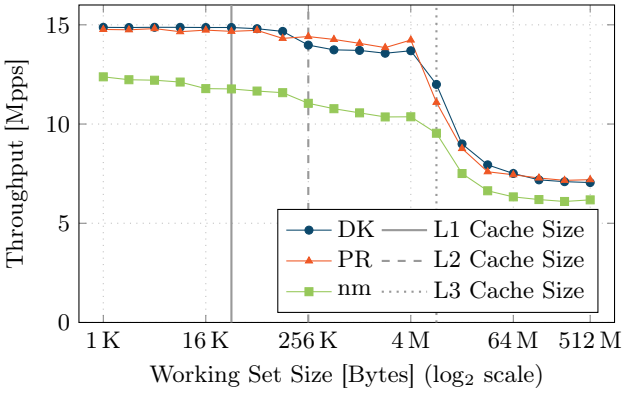
The CPU in our test server has 3 cache levels: A L1-cache with 32 KB, a L2-cache with 256 KB and a L3-cache with 8 MB [42]. Measurements showed that the average access time is 10 cycles for list sizes $\leq$ 32 KB, growing to 20 cycles for list sizes $\leq$ 256 KB, growing to 60 cycles for list sizes $\leq$ 8 MB, and finally reaching 250 cycles for list sizes larger than that.

The graph in Figure 3(a) shows no clear transition from L1 to L2 due to the low 10 cycle increase. The decline at around 256 KB is visible due to the larger speed difference between L2-cache and L3-cache. The next drop in the graph is the transition between L3-cache and non-cached RAM accesses.

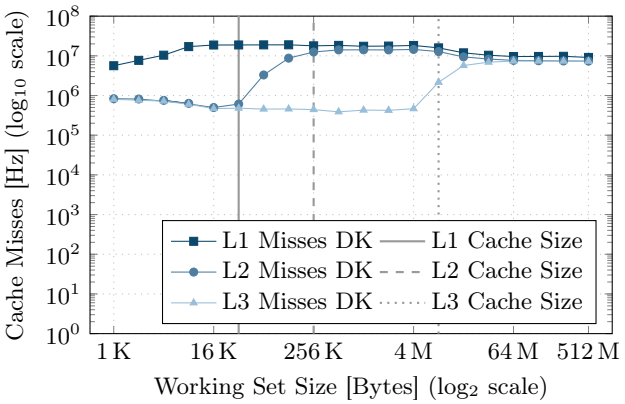
DPDK is slightly slower than PF_RING ZC when the L2-cache is fully occupied by the data structure. This means that DPDK has a slightly higher cache footprint compared to PF_RING ZC.

Figure 3(b) plots the cache misses, obtained by reading the CPU's performance registers. Only the results for DPDK are given. The results for netmap and PF_RING ZC are similar and are not shown for improved readability of the graph. The number of cache misses starts at a certain level and begins to rise as a cache fills up until the size of the test data exceeds the respective cache size. This observation holds for every cache level.

The data shown by Figure 3(a) can be used to test our model against a different problem. In contrast to the previ-



(a) Performance with memory accesses



(b) Cache misses for DPDK

Figure 3: Cache measurements

ously fixed load per packet, in this experiment the load per packet was determined by the cache access times. However, even under these circumstances the model provides a good estimation if the average per-packet costs are used. For instance, at a list size of 256 MB c_{task} the average costs to access a list element are 250 cycles. Taking the 100 extra cycles into account, this leads to average costs of 350 cycles for c_{task}. For DPDK the c_{IO} is roughly 100 cycles and f^{CPU} is 3.3 GHz. The expected throughput is 7.3 Mpps with our model, and the measured value in Figure 3(a) is 7.1 Mpps. The minor difference can be explained by the fact that the test data structure also competes for cache space with data required by the framework, which results in additional overhead beyond the cache miss when sending or receiving packets. Therefore, the size of data structures required for routing also needs to be considered when designing a software router.

5.5 Influence of Batch Sizes

In the following measurements, we analyze the influence of the batch size, i.e. the number of packets handled by one API call.

The tests shown in Figure 4 were conducted using different queue sizes with increasing CPU load using the task emulator. For each iteration of the test, the batch size was doubled starting at a batch size of 8 up to a batch size of 256. The results show that each framework profits from

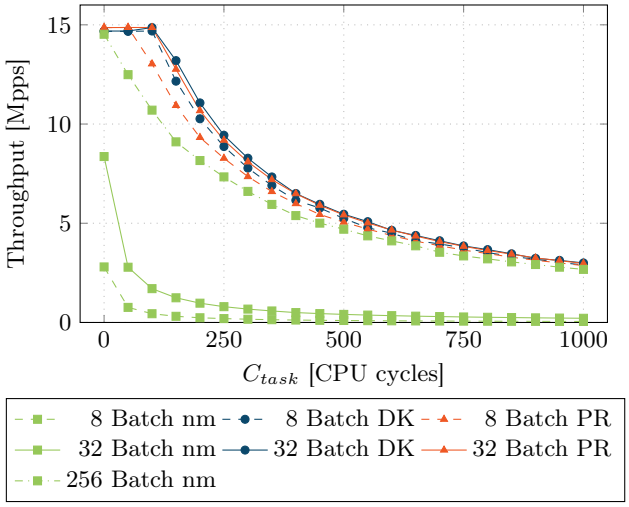


Figure 4: Throughput influenced by batch sizes

larger batch sizes. PF_RING and DPDK reach their highest throughput at a batch size of 32. The larger batch sizes are therefore omitted for those frameworks in Figure 4 because they also do not have adverse effects on the throughput. netmap needs a batch size of at least 256 to reach a throughput performance close to the other two frameworks. This is due to the relatively expensive system calls required to send or receive a batch (cf. Section 2.3).

5.6 Latency

Increasing the batch size boosts throughput but raises latency because the packets spend a longer time queued if processed in larger batches. Overloading a software forwarding application causes a worst-case behavior for the latency because all queues will fill up. So a high latency is expected for all cases where packets are dropped due to insufficient processing resources.

We used the IEEE 1588 hardware time stamping features of the Intel 82599 controller to measure the latency of the forwarding applications [9]. The packets are time stamped in hardware on the source and sink immediately before sending and after receiving them from the physical layer. The time stamps do not include any software latency or queuing delays on the source and sink. This achieves sub-microsecond accuracy. [39]

Figure 5 shows the latency for different batch sizes under a packet rate of 99% of the line rate[1] and no additional workload. The latencies were acquired by sending time stamped packets periodically (up to 350 time stamped packets per second) at randomized intervals by using a different transmit queue on the load generator. The time stamped packets are indistinguishable from the normal load packets for the forwarding application.

Both DPDK and PF_RING ZC are overloaded with a batch size of 8, netmap with all batch sizes smaller than 256 as described in the previous section. This causes all queues to fill up and the applications exhibit a worst-case behavior

[1]Using full line rate with constant bit rate traffic causes delays after a minor interruption (like printing statistics) because it is not possible to send faster than the incoming traffic.

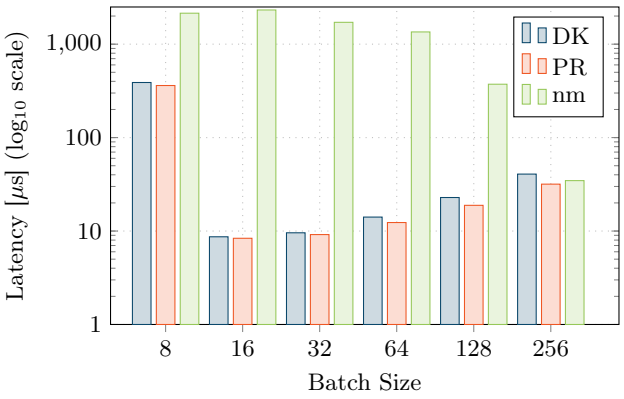


Figure 5: Latency by batch size

that is typical for a system that is overloaded. DPDK und PF_RING achieve a latency of 9 μs with a batch size of 16 and the latency then gradually increases with the batch size. PF_RING ZC gets slightly faster than DPDK for larger batch sizes. netmap achieves a forwarding latency of 34 μs with a batch size of 256.

These latencies can be compared to other forwarding methods and hardware switches: Rotsos et al. measured a latency of 35 μs for Open vSwitch under light load and 3 - 6 μs for hardware-based switches [32]. Bolla and Bruschi measured $\sim$15 μs to $\sim$80 μs for the Linux router in various scenarios without packet loss and latencies in the order of 1000 μs for overload scenarios [13].

6. CONCLUSIONS AND OUTLOOK

High-speed packet IO frameworks are no longer in fledgling stages and allow for a multiple of the packet rates of classical network stacks. The performance increase comes from processing in batches, preallocated buffers, and avoiding costly interrupts.

We described the processing performance of high-speed packet IO frameworks. Starting with a model describing packet processing software in general, this model is gradually adapted to reflect applications using high-performance frameworks. For our experiments we rely on a precisly generated load on the CPU. The code generating these different kinds of load on the CPU is publicly available [41]. Experiments showed the performance characteristics predicted by this model. Thus proving the assumptions right made during the development of this model. The CPU time spent on receiving and transmitting packets, for instance, remained constant despite the influence of the varying processing times per packet. Further measurements showed that this model can be applied to estimate processing tasks, which can be approximated with a constant average load. A possible use case for this model is to evaluate the eligibility of PC systems for specific packet processing tasks.

We also showed the trade-off between throughput and latency with different queue sizes. Larger batch sizes increase the performance but also the average latency. However, there is also a minimal batch size where the frameworks are overloaded. In that case latency is a multiple of what it could be if the packets would be sent in larger batches. These results can be used to choose the configuration and the framework best fit for an application's requirements, i.e. smaller batch sizes for applications sensitive to high latency or larger batch sizes for applications where raw performance is critical.

If mere performance and latency figures are considered DPDK and PF_RING ZC seem to be superior to netmap. Though netmap has advantages. It uses well-known OS interfaces and modified system calls for packet IO, leading to increased performance while remaining a certain degree of interface continuity and system robustness by performing checks on the user-provided packet buffers. DPDK and PF_RING ZC favor more radical approaches by breaking with those concepts, resulting in even higher performance gains, but lack the robustness and familiarity of the API. An application built on DPDK of PF_RING ZC can crash the system by misconfiguring the NIC, a scenario that is prevented by netmap's kernel driver.

Our conclusion is that the modification of the classical design for system interfaces results in higher performance. The more these interfaces are modified, the higher the packet rates that can be achieved. As a drawback, this requires applications to be ported to one of these frameworks.

Acknowledgment

This research was supported by the DFG as part of the MEMPHIS project (CA 595/5-2), the KIC EIT ICT Labs on SDN, and the BMBF under EUREKA-Project SASER (01BP12300A). We also want to express our sincere thanks to the anonymous reviewers, especially reviewer #1, for the constructive feedback.

7. REFERENCES

[1] L. Rizzo, "netmap: a novel framework for fast packet I/O," in *USENIX Annual Technical Conference*, April 2012.

[2] "Impressive Packet Processing Performance Enables Greater Workload Consolidation," in *Intel Solution Brief*. Intel Corporation, 2013, Whitepaper.

[3] "PF_RING ZC," http://www.ntop.org/products/pf_ring/pf_ring-zc-zero-copy/, last visited 2015-03-31.

[4] L. Rizzo and G. Lettieri, "VALE, a switched ethernet for virtual machines," in *CoNEXT*. ACM, 2012, pp. 61–72.

[5] "Intel Open Source Technology Center," https://01.org/packet-processing, last visited 2015-03-31.

[6] "netmap-ipfw," https://code.google.com/p/netmap-ipfw/, last visited 2015-03-31.

[7] F. Fusco and L. Deri, "High Speed Network Traffic Analysis with Commodity Multi-core Systems," in *Internet Measurement Conference*, November 2010, pp. 218–224.

[8] S. Gallenmüller, "Comparison of Memory Mapping Techniques for High-Speed Packet Processing," http://www.net.in.tum.de/fileadmin/bibtex/publications/theses/2014-gallenmueller-high-speed-packet-processing.pdf, 2014, last visited 2015-03-31.

[9] "Intel 82599 10 GbE Controller Datasheet Rev. 2.76." Intel Corporation, 2012.

[10] "Intel I/O Acceleration Technology," http://www.intel.com/content/www/us/en/wireless-network/accel-technology.html, Intel, last visited 2015-03-31.

[11] J. H. Salim, "When NAPI Comes To Town," in *Linux 2005 Conf*, 2005.

[12] M. Dobrescu, K. Argyraki, and S. Ratnasamy, "Toward predictable performance in software packet-processing platforms," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 11–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228313

[13] R. Bolla and R. Bruschi, "Linux Software Router: Data Plane Optimization and Performance Evaluation," *Journal of Networks*, vol. 2, no. 3, pp. 6–17, June 2007.

[14] J. L. García-Dorado, F. Mata, J. Ramos, P. M. Santiago del Río, V. Moreno, and J. Aracil, "Data Traffic Monitoring and Analysis," E. Biersack, C. Callegari, and M. Matijasevic, Eds. Berlin, Heidelberg: Springer-Verlag, 2013, ch. High-Performance Network Traffic Processing Systems Using Commodity Hardware, pp. 3–27.

[15] L. Deri, "nCap: Wire-speed Packet Capture and Transmission," in *End-to-End Monitoring Techniques and Services*. IEEE, 2005, pp. 47–55.

[16] "netmap(4)," in *FreeBSD 11 Man Pages*. The FreeBSD Project, 2014.

[17] S. Hemminger, "netmap: infrastructure (in staging)," http://lwn.net/Articles/548077/, last visited 2015-03-31.

[18] "The Click Modular Router Project," http://www.read.cs.ucla.edu/click/, last visited 2015-03-31.

[19] "PF_RING ZC API," http://www.ntop.org/pfring_api/pfring__zc_8h.html, last visited 2015-03-31.

[20] "Ntop," http://www.ntop.org, last visited 2015-03-31.

[21] "Data Plane Development Kit: Programmer's Guide, Revision 6." Intel Corporation, 2014.

[22] "UIO: user-space drivers," http://lwn.net/Articles/232575/, last visited 2015-03-31.

[23] "xDPd," https://www.xdpd.org, last visited 2015-03-31.

[24] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: a GPU-accelerated Software Router," *SIGCOMM Computer Communication Review*, vol. 40, no. 4, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=2043164.1851207

[25] "PacketShader Packet-IO-Engine," https://github.com/PacketShader/Packet-IO-Engine, last visited 2015-03-31.

[26] N. Bonelli, A. Di Pietro, S. Giordano, and G. Procissi, "On Multi–Gigabit Packet Capturing With Multi–Core Commodity Hardware," in *Passive and Active Measurement*. Springer, 2012, pp. 64–73.

[27] N. Bonelli, S. Giordano, G. Procissi, and L. Abeni, "A purely functional approach to packet processing," in *Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '14. New York, NY, USA: ACM, 2014, pp. 219–230.

[28] "Snabb Switch: Fast open source packet processing," https://github.com/SnabbCo/snabbswitch, last visited 2015-03-31.

[29] G. Pongracz, L. Molnar, and Z. L. Kis, "Removing Roadblocks from SDN: OpenFlow Software Switch Performance on Intel DPDK," *Second European Workshop on Software Defined Networks (EWSDN'13)*, pp. 62–67, 2013.

[30] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, "Assessing Soft- and Hardware Bottlenecks in PC-based Packet Forwarding Systems," in *Fourteenth International Conference on Networks (ICN 2015)*, Barcelona, Spain, Apr. 2015.

[31] A. Tedesco, G. Ventre, L. Angrisani, and L. Peluso, "Measurement of Processing and Queuing Delays Introduced by a Software Router in a Single-Hop Network," in *IEEE Instrumentation and Measurement Technology Conference*, May 2005, pp. 1797–1802.

[32] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "Oflops: An Open Framework for OpenFlow Switch Evaluation," in *Passive and Active Measurement*. Springer, 2012, pp. 85–95.

[33] S. Larsen, P. Sarangam, R. Huggahalli, and S. Kulkarni, "Architectural Breakdown of End-to-End Latency in a TCP/IP Network," *International Journal of Parallel Programming*, vol. 37, no. 6, pp. 556–571, 2009.

[34] PCI-SIG, "Express base specification revision 2.0," 2006.

[35] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Elsevier, 2012.

[36] L. Braun, C. Diekmann, N. Kammenhuber, and G. Carle, "Adaptive Load-Aware Sampling for Network Monitoring on Multicore Commodity Hardware," in *IFIP Networking*, May 2013.

[37] "netmap," https://code.google.com/p/netmap/, last visited 2015-03-31.

[38] "DPDK," http://www.dpdk.org, last visited 2015-03-31.

[39] P. Emmerich, S. Gallenmüller, F. Wohlfart, D. Raumer, and G. Carle, "MoonGen: A Scriptable High-Speed Packet Generator," http://go.tum.de/276657, 2015, *Draft, Conference tbd.*

[40] G. Paoloni, "How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures," 2010.

[41] "Simulation algorithms for empirical evaluation of processor performance," https://github.com/gallenmu/sheep, last visited 2015-05-28.

[42] "Intel 64 and IA-32 Architectures Optimization Reference Manual." Intel Corporation, 2012.