Pktgen: measuring performance on high speed networks

Daniel Turull[a,*], Peter Sjödin[a], Robert Olsson[a]

[a]*KTH Royal Institute of Technology, School of ICT, Forum 120, 164 40 Stockholm, Sweden*

Abstract

Pktgen is a tool for high-speed packet generation and testing. It runs in the Linux kernel, and is designed to accommodate a wide range of network performance tests. Pktgen consists of a packet generator, a receiver, and a protocol that defines the format for test packets. This paper describes the design of pktgen, and discusses its usage as a capable performance testing tool. The design is focused on performance, and in order to generate packets at high packet rate, pktgen takes advantage of multicore systems and multi-queue features on modern network cards. Pktgen supports generation of UDP flows, to allow creating realistic heterogeneous traffic patterns.

In this paper, we evaluate and compare pktgen to other tools, and show that it is more than an order of magnitude faster than current Linux applications such as Iperf and Netperf, and has the same upper limit on performance as special-purpose high-speed tools, such as DPDK and Netmap, when it comes to throughput measurements with user-specified rate. Pktgen supports a wide-range of fine-grained user specified rates. Furthermore, pktgen supports latency measurements with rate control, and our experiments indicate that pktgen can be used for latency measurements with high resolution.

Keywords: Pktgen, packet generator, Linux, network performance, measurement

*corresponding author: telephone: +46 73 095 82 90. Permanent address: Ericsson Research, Färögatan 6, 164 80 Stockholm, Sweden

Email addresses: danieltt@kth.se (Daniel Turull), psj@kth.se (Peter Sjödin), roolss@kth.se (Robert Olsson)

1. Introduction

Network performance testing is crucial for the development of network systems, services and protocols. It is an active measurement method where a stream of packets is sent over the network and measured in order to determine, for example, throughput and latency. Network performance testing can be used for many different purposes, such as, network troubleshooting, characterizing performance properties of switches and routers, software development, and network security analysis.

Commercially available equipment for network performance testing is often based on costly dedicated hardware and uses proprietary software for traffic generation and analysis. Alternatively, there are open source applications that allow regular computers to be used for network performance testing, such as Netperf [1] and Iperf [2]. They have advantages in terms of usability and flexibility, but suffer in performance due to the overhead incurred by running as applications in user space on top of the standard socket API (application programming interface).

The purpose of this paper is to present pktgen, a Linux open-source software package intended to fill the gap in performance between dedicated hardware equipment and open-source applications. Our approach to this is twofold: First, we move the performance-critical parts into kernel space, making it possible to perform packet processing close to the hardware with very little overhead. Second, we design pktgen to take advantage of recent advances in computer architectures in order to run as efficiently as possible. In particular, pktgen exploits the support for parallelism that is available through multiple processor cores and network interface cards with multiple queues.

Linux offers significant advantages for testing purposes. Its source code is open, which permits us to modify and adapt it to our needs. Additionally, it has been ported to more computer platforms than any other operating system, and runs on powerful servers as well as on small embedded devices. Also, it has a wide range of network drivers and a large community to support it.

Pktgen has three main parts: The first is the packet generator [3], which sends packets on a network interface. The packet generator is part of the standard Linux kernel distribution since 2002, and has been revised and refined several times since then. The second part is the receiver, a more recent addition for collecting receiver statistics. The third part is the pktgen

protocol, which defines the format for test packets, including timestamps and sequence numbers, and was developed together with the receiver. Pktgen has been verified to run on Intel and ARM platforms. The generator was initially created by one of the authors of this paper, and has been extended and maintained by the Linux networking community, while the receiver is not currently part of the current Linux kernel distribution.

Pktgen is designed to be a versatile tool, which we believe can be useful for a wide range of applications: for instance, analysis and improvements of open source routers and other networking equipment; development of drivers for network cards and other system-level software; troubleshooting networks by studying them under varying traffic conditions; performance benchmarking of routers and switches; network security analysis by investigating resistance to denial-of-service attacks, and so on.

The rest of the paper is organized as follows: Section 2 describes the architecture of pktgen. Implementation aspects of pktgen are discussed in Section 3, with focus on performance. Section 4 reviews related tools and discusses other approaches to network performance testing. Section 5 starts with a more general discussion about software tools for performance evaluation and the implications for the quality of the measurements. Thereafter, Section 5 discusses the experimental setup and provides some guidelines for configuring a Linux system for performance measurements, and presents measurement results in terms of throughput, latency and jitter. Finally, Section 6 concludes the paper.

2. Architecture

Pktgen is designed for performance testing that involves sending and receiving packets at high rate, while still being general enough to run on any Linux system. To accomplish this, the main objectives in the design are:

- Implement pktgen with as little overhead as possible, in order to generate and receive packets at high rate.
- Use existing driver structures without modifications, to enable pktgen to run on any network card with a Linux driver.
- Support for multiple UDP flows. This allows for a wider range of traffic patterns, which is useful for testing forwarding capabilities of network equipment.

- Gather measurement data to support standard benchmarking tests, such as, throughput, latency and jitter.

Pktgen consists of a Linux kernel module with three parts, as illustrated in Figure 1: generator, receiver and a protocol. In addition, there is a user interface where the user can control pktgen through read and write operations to the `/proc` file system.

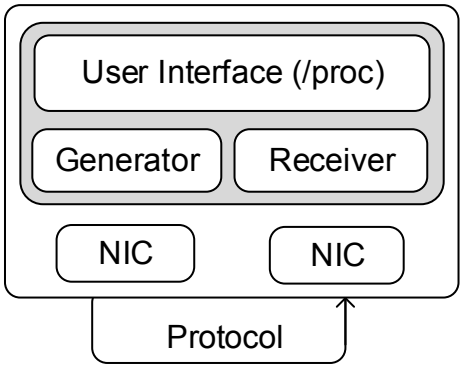

Figure 1: The main components of the pktgen architecture. Pktgen consists of a generator that sends packet via a network interface card (NIC); a receiver that receives packet from a NIC, and collect receiver statistics; and a protocol that defines the format for generated packets. Pktgen is controlled through a user interface built on top of the `/proc` file system.

2.1. Pktgen protocol

Pktgen packets are transmitted as UDP/IP datagrams. Each packet contains a pktgen header in the UDP payload, which permits to identify the packets as pktgen packets by the receiver. By default, pktgen uses the discard port (port 9) for UDP, but this is configurable. Pktgen supports IP version 4 and version 6.

The pktgen header, illustrated in Figure 2, has the following fields: an *identifier* showing that the packet was created by pktgen (4 bytes), a *sequence number* increased by one for each packet (4 bytes), and a *timestamp* with nanosecond resolution indicating when the packet was created (8 bytes). The timestamp is for latency calculations.

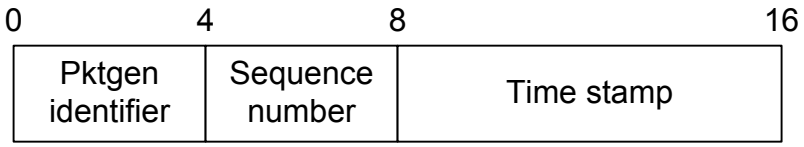

Figure 2: The pktgen packet header. The identifier is a 4-byte word identifying the packet as a pktgen packet. The sequence number starts from zero, and is incremented by one for each packet in the test. The 8-byte timestamp has nanosecond resolution and is intended for latency measurements.

2.2. Generator

The pktgen generator is configured through a user interface, where the user specifies the parameters for the test. The overall operation of the generator is straightforward: When the test is running, the generator creates packets according to the test parameters, and calls the driver's function for sending the packets on the network.

Figure 3 shows the generator design. Pktgen can have several independent generators, each responsible for sending packets on one interface (or, more specifically, one network interface queue). Each generator is controlled by a thread. Since pktgen runs in the kernel, it directly invokes the driver function for sending packets, rather than invoking it indirectly through the socket API. Furthermore, the pktgen generator can reuse the same packet buffer for multiple pktgen packets, thereby reducing the number of buffer management operations, which in turn decreases latency and increases overall performance.

2.3. Receiver

To keep the amount of processing down, the pktgen receiver should be located as early as possible in the protocol processing chain. Therefore, we have implemented the receiver as a module called at a Netfilter *hook*. Netfilter[1] is a framework in the Linux kernel that enables packet filtering and mangling. It allows packet processing functions to be inserted at well-defined points, or hooks, in the network stack. The hook NF_INET_PRE_ROUTING comes early in the packet processing chain, right after IP header validation and before destination address lookup. We introduce our receiver there, as

[1] `http://www.netfilter.org/`

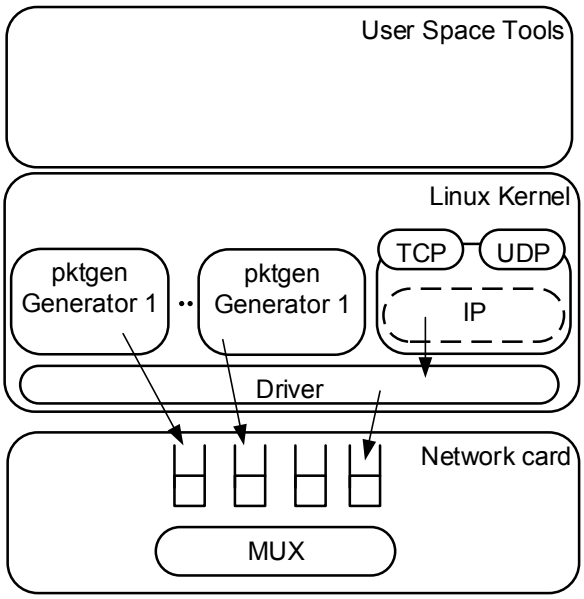

Figure 3: The generator architecture. A number of packet generators running concurrently, each controlled by a kernel thread. The generators output packets to network interface queues, from which the packets are multiplexed onto the network. Two user-space tools are also shown, Netperf and Iperf, which send packets through the socket API.

shown in Figure 4. The receiver checks if the packet is a pktgen packet; if so, receiver statistics are updated and the packet is dropped.

The alternative to using Netfilter hooks would be to have the receiver as an IP protocol handler, in the same way as for TCP and UDP. In that case, pktgen packets would travel much further up the protocol stack, which in turn would increase CPU usage. One might also consider having pktgen as a network layer protocol alongside IP, with the receiver being called directly from the network interface at the link layer. That would eliminate the processing for IP header validation, but at the expense of the generality that comes from having pktgen as a protocol carried by IP.

2.3.1. Reception measurements

The receiver collects reception measurement statistics. We have selected some of the measurements from the IETF recommendations in RFC 1242, RFC 2544, and RFC 4689. These RFCs define the methodology and terminology for benchmarking *network interconnect devices*. Pktgen supports a set of fundamental measurements, which can be combined into more complex

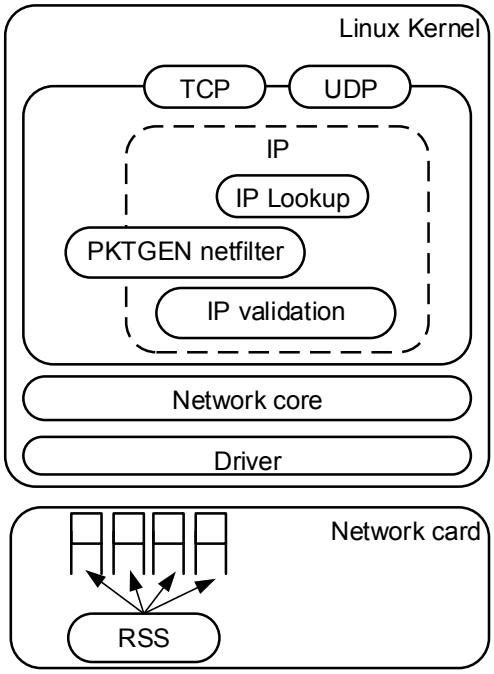

Figure 4: The receiver architecture. The pktgen receiver is called at a Netfilter hook in the IP stack, right after header validation.

measurements. The supported measurements are: throughput, inter-arrival time, latency and packet delay variation. A brief explanation can be found in Table 1.

For the inter-arrival time, latency and packet delay variation measurements, measurement data are reported as average, variance, maximum and minimum values, represented as 64-bit unsigned integers. In order to do this, the receiver maintains a five-tuple $\langle N, x^{min}, x^{max}, \sum_i x_i, \sum_i x_i^2 \rangle$, which is updated for each new sample. Here, N denotes the number of samples from the start of the test, x^{min} the minimum sample, x^{max} the maximum sample, $\sum_i x_i$ the accumulated sum of the samples, and $\sum_i x_i^2$ the accumulated sum of the squares of the samples. When the test is finished, the average is obtained as $\sum_i x_i/N$ and variance as $\sum_i x_i^2/N - (\sum_i x_i/N)^2$.

The notion of packet delay variation (or jitter) is not entirely well-defined, and there are alternative ways of measuring it (RFC 3393 or RFC 4689,

Measurement	Explanation
Throughput (packet rate)	Amount of data received per second (packets per second and bits per second)
Inter-arrival time	Difference in arrival time between two consecutive packets (nanoseconds)
Latency (packet delay)	Time from that the packet is sent by the generator, until the packet is received by the receiver process (nanoseconds)
Packet delay variation (jitter)	Absolute difference in latency between two consecutive packets (nanoseconds)

Table 1: Measurements supported by the pktgen receiver. Inter-arrival time, latency and packet delay variation data are reported as average, variance, maximum and minimum values. All values are represented as 64-bit unsigned integers.

for instance). Pktgen calculates packet delay variation during the latency measurements as the absolute difference in latency between two consecutive packets, as described to RFC 4689. Another possibility, according to RFC 3393, would be to take the difference in latency between the packets with maximum and minimum latency during the entire test. Since both those values are included in the results of a latency measurement, packet delay variation according to this alternative definition could be also be obtained directly from a latency measurement.

3. Implementation

Pktgen is implemented as a loadable kernel module. This section analyses key components in the implementation, focusing on performance-related aspects.

3.1. Timers

Pktgen measurements involve timekeeping. For this, pktgen relies on the internal clock of the local system. More specifically, it uses the kernel function `ktime_get()` to obtain timestamps. This is a generic function that uses the best timer available. On current Intel-based platforms, this usually means HPET, the High Precision Event Timer. It is a timer with nanosecond resolution, and with clock frequency drift of at most 500 ppm (parts per million) over intervals longer than one millisecond. For shorter intervals, the drift may be higher, up to 2000 ppm [4].

3.2. Transmission granularity

The pktgen generator allows a constant packet rate to be assigned to each flow, expressed either in bits per second or packets per second. The rate is controlled by adjusting the inter-packet delay. Pktgen has two different ways of doing this: active or passive waiting. Active waiting consists of a busy-wait loop while passive waiting reschedules the thread and yields the CPU. The choice of which waiting method to use is made based on a threshold value, spin time, which by default is 0.1 milliseconds. If the remaining time before the next packet should be sent is less than the spin time, active waiting is used, otherwise passive. Active waiting is more accurate, but blocks the CPU. Both methods are necessary in order to achieve a wide range of packet transmission rates.

With the HPET timer, the pktgen generator can get granular control of packet rate, but at a certain price. High resolution timers take slightly longer time to read compared to system timers with lower resolution. Hence, checking the timer too often could decrease the total throughput.

3.3. Multi-queue support

Modern network chipsets have hardware support for multiple receive and transmit queues [5]. On a multicore system, receive and transmit queues can be assigned to CPUs so as to allow simultaneous accesses to different queues. Furthermore, a queue can be assigned to an interrupt handler, which in turn is bound to a CPU through CPU affinity [6]. In this way, it will always be the same CPU that serves a given queue, which is advantageous for cache performance. In combination with Receive Side Scaling (RSS), which aims to direct packets belonging to the same flow to the same queue, packet flows will be tied to CPUs – all packets in a flow will go to the same CPU. This increases the probability of cache hits, which in turn increases performance [5].

3.3.1. Transmission

Pktgen allows the user to specify which transmit queue to use, which allows pktgen to be configured to have multiple CPUs transmitting simultaneously on the same network interface. By parallelizing transmissions in this way, it is possible to achieve a higher packet rate compared to only using a single CPU.

3.3.2. Reception

Pktgen reception is optimized for network interfaces with multi-queue features, by letting each core have its own set of data structures for receiver processing. This allows fast access to those data structures, by eliminating the cost of managing simultaneous accesses to shared resources. Furthermore, a pktgen packet is dropped as early as possible – once the pktgen netfilter module has processed it – in order to minimize the resources consumed by receiver processing. The timestamps that are needed for the receiver measurements are obtained in the netfilter module as soon as the function is called. One might argue that this will not give the most accurate representation of reception time, since there is a certain delay from that a packet is received on an interface until it reaches the netfilter module. However, it is a generic approach, and it does not depend on specific hardware.

3.4. Memory node awareness in NUMA systems

Multiprocessor systems often have memory organized in a Non-Uniform Memory Access (NUMA) design. In such a design, there are multiple banks of memory. Each processor is physically connected to one memory bank – the local memory for that processor. Consequently, it is faster to access local memory than non-local memory. In addition, computers often have multiple internal bus systems, and the time to access a network card depends on the bus to which the card is connected. Hence, the access time depends on the location of the network card, the memory bank, and the processor.

Hagsand et al. [7] use pktgen to generate traffic in experiments to study network performance on NUMA systems, and demonstrate that forwarding performance depends on the memory bank in which packets are stored. Moreaud et al. [8] also show the impact of NUMA on high-speed networking. To optimize performance on NUMA systems, pktgen includes an option through which the user can specify the memory bank where packet buffers should be allocated.

3.5. Flow support

Pktgen provides the possibility to create multiple concurrent UDP flows. This is intended to be used, for instance, to test forwarding elements, generate background traffic, or mimic applications that send streams of UDP data, for instance real-time video. Network traffic is heterogeneous in nature and comes from many sources, and test traffic with static patterns may therefore give less representative results, for instance, for routers that use caching to

improve performance. Therefore, pktgen supports *flows* as a way to generate traffic with varying traffic patterns.

Pktgen has the possibility to send multiple concurrent flows, each with a specified length. Within a flow, all packets have the same IP address and UDP port number. Once a flow is finished, a new is created. Multiplexing between flows is done randomly or sequentially. With random multiplexing, the next packet to send is chosen randomly from one of the active flows, while with sequential multiplexing, a new flow is started after finishing the current flow.

3.6. Bulk transmission support

A recent addition to the Linux kernel introduces support for *transmit bulking* [9][2]. The operation to notify the hardware that a packet is ready to be sent is expensive, in terms of latency and cache utilization. Bulking allows this cost to be amortized over several packets, by deferring the notification in the case that the user has more packets to send. The pktgen generator supports bulking, and Brouer et al. report wire-speed performance with small packets (64 bytes) at 10 Gb/s on a single CPU core [9].

4. Other network performance tools

Currently there are several software tools that can be used to analyze network performance using commodity hardware. They run on general-purpose systems, and some of them are open-source.

Netperf [1] is a de facto standard tool for testing network performance in IP networks. It runs in user space and uses the standard socket API, and therefore has limited access to the hardware. Netperf can be used to test bidirectional throughput and end-to-end latency over UDP, TCP and SCTP. Iperf [2] is another user-space tool that permits to measure TCP and UDP throughput, delay, jitter and packet loss between two hosts. User-space tools such as Iperf and Netperf are flexible, easy to deploy and permit a wide variety of traffic patterns. Their main drawback is low performance, due to the overhead involved in invoking network drivers indirectly through the socket API, which limits the tools' usefulness on high-speed networks. Other examples of user space tools include Ostinato [10], mgen [11], Rude/Crude [12], D-ITG [13] and ttcp [14].

[2]The Linux kernel supports transmit bulking as of version 3.18.

Network stacks in operating systems have been growing in terms of network functionality, resulting in higher complexity and larger data structures, which could potentially hamper performance. This leads the authors of Netmap [15] to propose a new architecture to increase network performance. The idea behind Netmap is to reduce the size of packet data structures and give applications direct access to the internal memory of the network card. Netmap has its own API, through which applications get direct access to packet buffers on network cards. This effectively bypasses packet processing in the operating system, and thereby increases performance.

A consequence of this design is that protocol processing is moved from operating system kernel into the applications, which therefore need to be designed specifically for Netmap. Even though Netmap can also run with regular network drivers, it requires driver modifications in order to achieve maximum performance. The Netmap toolset includes a sample packet generator application, which we use for our comparisons. However, Netmap is not mainly intended as a performance testing tool. Its purpose it to provide a framework for fast packet processing by allowing applications to circumvent the operating system and access network cards directly.

A similar approach is taken with Intel's Data Plane Development Kit (DPDK) [16]. DPDK is an open-source library that permits to process packets in user space by directly accessing the network card. Like Netmap, DPDK bypasses the network drivers in the operating system. It detaches the network interfaces from the operating system, and therefore the network interfaces can only be accessed through the DPDK library. DPDK only supports a limited set of NICs (mostly Intel, but other vendors support DPDK too). In contrast, Netmap permits to manage network interfaces with operating system tools. The DPDK platform also includes a sample packet generator application, which we use for our experiments.

Unfortunately there is a slight name clash among pktgen, Netmap and DPDK. The Netmap sample packet generator application is called "pkt-gen" while the DPDK packet generator is called "Pktgen", or sometimes "Pktgen-DPDK". To avoid name confusion, we take the liberty of using "pktgen" exclusively for the Linux kernel tool, and refer to the others by the names of their frameworks (that is, "DPDK" and "Netmap").

There are other high-performance packet generators reported in the literature, which we do not include in this study, for different reasons. Bonelli et al. [17] present an architecture for an accelerated transmission socket, which also takes advantage of multi-queue and multi-core architectures. Its

source code is not available, therefore we are not able to evaluate it. BRUTE [18] is a kernel packet generator with a modular architecture, but it is not designed to take advantage of parallelism and other features of modern computers, and therefore it is not included here. KUTE [19] is a UDP traffic generator designed for gigabit interfaces. Its reported performance is relatively low, and there appear to be no further development activities. Schneider et al. [20] modified a previous version of pktgen by adding the capability to vary packet size and packet rate during a transfer. This modification is not included in the Linux kernel distribution.

The main focus of this paper is on software tools. However, there are also proprietary hardware tools commercially available for network performance testing, such as, Ixia's IxNetwork [21] and Spirent's SmartBits [22]. These tools are in general more precise and accurate than software tools, and reach better performance. The drawback is that they are more expensive and often provide less flexibility, compared to software-based tools.

5. Running experiments with pktgen

In this section, we demonstrate how pktgen can be used as a network analysis tool by running a number of experiments on a test system and discussing the results. We also compare pktgen with other tools. Before doing that, we discuss the usage of a general purpose operating system on commodity hardware for performance measurements, and provide guidelines for system configuration.

The experiments consist of first running three different throughput measurements obtaining the receiver metrics presented in Section 2.3. Then we investigate latency and packet delay variation measurements, and conclude with a discussion of the results.

5.1. Measurement quality

Consider a measurement as the process of using a measurement system to obtain a measured value as a representation of a physical quantity, a real value. Then there will be an error in the measurement – a difference between the measured value and the real value – since measurement systems have imperfections and are exposed to disturbances. In metrology, the following terminology is often used to describe errors in measurements [23]:

- **Resolution:** The smallest change in real values that can be measured.

- **Accuracy**: How close a measured value is to the real value.
- **Precision:** How spread out measured values are when repeating measurements of the same real value.

For a hardware packet analyzer, its resolution, accuracy and precision could be expected to be well-known and given in the specification of the measurement device. Unfortunately, this is not easily done for software measurement tools. There is a plethora of possible hardware configurations on which the tools can run, and the performance characteristics of such systems tend to be imprecise and hard to determine. Botta et al. [24] find in their study that the lack of precision and accuracy with general purpose operating systems can significantly impact the quality of experimental results. They also show that it is a complex task to evaluate performance of individual functions, since several tasks compete for hardware resources.

5.2. System Configuration

For experiments that consists of several different tests, we need to ensure that the tests run under as similar conditions as possible, so that we can compare the results. For a Linux system running on a contemporary Intel-based platform, there are several configuration options that could influence the precision of the measurements. Below we describe some important configuration options and explain the settings that we use in our experiments:

- Frequency scaling dynamically adjusts CPU frequency as the workload varies. We disable frequency scaling and set a fixed CPU frequency.
- Power saving modes ("C-states") have certain wake-up times when the CPU goes from sleep mode to fully active [25]. Deeper sleep modes have longer wake-up time, so we limit CPU power saving state to "C1," the state with shortest wake-up time.
- CPU affinity (or pinning) ties the execution of a process to a given CPU. We use this to fix pktgen processes to CPUs, which reduces the probability of cache misses.
- Using the multi-queue support available on the network interfaces, we bind transmit and receive queues to CPUs, as described in Sect 3.3. Together with CPU affinity, this ties pktgen processes to interface queues, and thereby reduces the probability of cache misses.

- Interrupt load balancing automatically dispatches interrupts to CPUs according to load, something that can affect locality of memory references and cache hit ratio. Therefore, we disable interrupt load balancing.

- NUMA affinity controls the memory bank in which packets are stored, as described in Sect. 3.4. Pktgen has support for pinning packet buffers to memory banks, which we use to make sure that packets are stored in the memory bank that gives best performance. (A certain amount of experimentation might be required prior to running the measurements in order to identify the best memory bank.)

- Ethernet flow control allows a saturated receiver to throttle frame rate by sending a Pause frame to the generator and thereby temporarily stop the transmission. We disable this feature.

- Adaptive Interrupt Moderation (IM) [26] adapts interrupt rate to incoming packet rate by delaying interrupts, waiting for more packets to arrive. This has the advantage of reducing CPU processing load and increasing bus efficiency, but at the expense of increased latency for some packets. Therefore, for latency measurements we disable interrupt moderation, while for throughput-related measurements we leave it on.

- Placing the sender and receiver in different machines avoids possible contention in system buses and CPUs, which helps when the measurement workload comes close to the limit of what a single system can handle. We use this configuration for throughput-related measurements, while latency measurements are made with the sender and receiver on the same machine (which simplifies synchronization).

5.3. Setup

The systems used for testing have the following configuration: motherboard TYAN S7002, Intel® Xeon® CPU E5520 at 2.27 GHz, which has four cores with two threads in each. 3 GB (3×1 GB) of DDR3 (1333 MHz) RAM. The network card is a Dual Fiber Port Intel® 82599EB 10 Gb/s Ethernet card. The operating system is Ubuntu 13.10 with Linux kernel version 3.18.0.

5.4. Throughput

Throughput, or packet rate, is an important measurement for performance testing. RFC 2544 defines throughput as the fastest rate at which packets can be forwarded without packet loss, which implies a measurement method with granular rate adjustments until packet loss occurs. Therefore, it is important that the tool provides the ability to accurately set the transmission rate. Throughput measurements are made at steady state load, in other words, with packets generated with constant inter-packet time. Furthermore, RFC 2544 specifies throughput to be measured as a function of packet size, and reported as packets (or frames) per second.

We perform three experiments related to throughput: maximal throughput, user-specified packet rate and parallelization. In the first and second experiment we use pktgen, Iperf, Netperf, Netmap and DPDK. For Netmap and DPDK, we use the sample applications provided with the distribution packages to generate and receive packets. The third experiment (parallelization) is performed only for pktgen, and involves sending multiple streams at the same time with different destination addresses.

All three throughput measurements use the following experiment configuration: two machines, generator and receiver, are directly connected with a cable. Each test runs for a minute, and uses UDP packets with 18 bytes of payload. With IPv4, this corresponds to the minimum Ethernet frame size of 64 bytes.

5.4.1. Maximal throughput

Netmap, DPDK and pktgen have support for *packet bursts*. A packet burst is a cluster of packets that are sent back-to-back without any delay in between. The size of the burst is user-configurable. Packet bursts are implemented by transferring multiple packets to the network interface in a single operation. Pktgen uses bulking in the Linux kernel to send packet bursts, as described in Section 3.6, while Netmap and DPDK rely on their own special-purpose drivers. In contrast, Iperf and Netperf do not have support for packet bursts.

Packet bursts is an efficient way of sending packets, since many packets can be sent with a single driver operation, thus reducing transmission overhead. However, it leads to bursty traffic, and hence cannot be used for measurements with constant packet rate.

In order to investigate maximal throughput, we measure receive and transmit rates for DPDK, Iperf, Netmap, Netperf, and pktgen without user-

Tool	TX Rate (kp/s)	RX Rate (kp/s)	Kernel driver
Iperf3	294	294	original
Netperf	713	713	original
Pktgen	8 100	2 060	original
Netmap	14 880	11 200	customized
DPDK	14 880	14 880	customized

Table 2: Packet rates as reported by sender and receiver for one-minute transfers of UDP packets in 64-byte Ethernet frames using one CPU core, with no user-specified rate given. Packet bursts are enabled for DPDK, Netmap and pktgen, with burst size 10.

specified rate, and with packet bursts enabled for DPDK, Netmap and pktgen, using a burst size of 10 packets. We also use `perf` [27] Linux kernel profiling for performance monitoring. The results are shown in Table 2.

We observe that the two user-space tools, Iperf and Netperf, have much lower performance. On the transmitter side, the `perf` profiling data indicates that in both cases, the bottleneck operation is the copying of data between user and kernel space. Iperf, Netperf, and pktgen have 100% CPU utilization. Therefore, we conclude that for those tools, packet generation with small packets is CPU-bound. Netmap saturates the link, but does not reach full CPU utilization, indicating that it does not run at its maximum speed. In contrast to the other tools, DPDK uses polling and runs in a busy-wait loop with 100% CPU utilization, regardless of load. Hence, the profiling data does not say much about DPDK's performance characteristics.

On the receiver side, the pktgen and Netmap receivers do not keep up with their transmitters. The profiling data show that in both cases, the CPU is not saturated, indicating that the bottleneck lies in the communication over the internal buses between CPU, memory and network adapter. In contrast, the DPDK receiver reaches wire-speed, using 100% CPU with its polling loop.

5.4.2. User-specified packet rate

In this experiment we investigate how the different tools are suited for RFC 2544 throughput tests, by measuring actual packet rate, as reported by the tools, as a function of user-specified rate. Iperf and Netperf are not multi-threaded, so in order to obtain comparable results we run this experiment on a single core configuration.

Throughput tests according to RFC 2544 are made as a serie of mea-

surements each at a specified, constant packet rate. A measurement is one minute long, as recommended by the RFC, and uses UDP packets in 64-byte Ethernet frames. Since packet rate should be constant, we disable packet bursts in DPDK, Netmap and pktgen.

Setting a user-specified rate incurs a certain overhead, which comes from the computations to ensure proper intervals between packets. Therefore, an important property of a tool when it comes to throughput measurements is the maximum user-specified packet rate that can be accurately achieved (that is, where actual rate equals user-specified rate).

Figure 5 shows the results of running throughput measurements with DPDK, Iperf, Netperf, Netmap and pktgen. The figure shows the maximum user-specified rate that can be achieved with each tool, as well as the rate that is obtained without user-specified rate. It can be seen that with Netperf, Netmap and pktgen, it is possible to achieve a significantly higher rate by not setting a user-specified rate, which serves to illustrate the overhead involved in limiting the rate.

The support for setting user-specified rate varies between the tools: Netperf only allows setting packet rate indirectly, by specifying an inter-packet delay. The minimal value is 1 millisecond, which puts an upper limit of 1 kp/s on user-specified rate. Iperf does not allow setting packet rate directly. Instead, the user can specify the throughput for UDP payload in bits per second. The DPDK packet generator application allows the user to specify the target rate expressed as a percentage of the maximum link speed (with a resolution of 1%), while with the Netmap and pktgen generators the user specifies the target rate as an absolute number (in packets per second).

5.4.3. Parallelization

We have designed pktgen to take advantage of parallelization and other modern features in current computer systems. In this experiment, we study the effects on performance of using parallelization, by measuring maximum transmit and receive rate while varying the number of generators and receivers, allocating one CPU core to each generator and receiver. The processor in the experimental setup has four CPU cores, each with two virtual cores (hyper-threading).

We run two measurements at maximal throughput configuration (no user-specified rate, and with 10-packet bursts). In the first measurement, we use the same number of cores for sending as for receiving, first with hyperthreading disabled, varying the number of cores from one to four, and then with

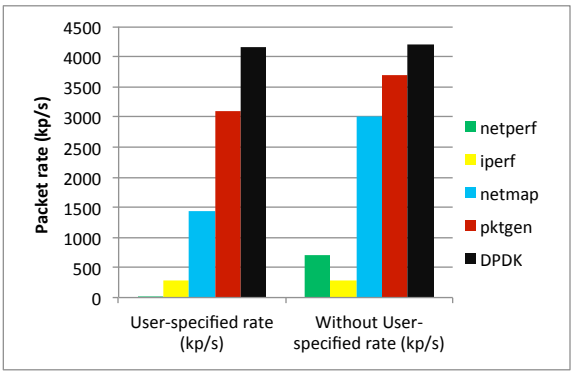

Figure 5: Analysis of user-specified rate. The diagram shows the maximum user-specified packet rate that is possible to achieve with each tool, as well the rate obtained when the user does not specify a rate. The packet rates are for one-minute transfers of UDP packets in 64-byte Ethernet frames using one CPU core. Packet bursts are disabled for DPDK, Netmap and pktgen. Netperf has a built-in limitation of 1 kp/s for user-specified rate, so it is not visible in the diagram.

hyperthreading activated (which gives eight cores total). We want to distribute the traffic evenly to the receivers, and for this we use the multiflow support in pktgen. Short-lived flows (four packets long) are generated with destination addresses randomly picked from a pool of 255 addresses. Thanks to the Receive Side Scaling (RSS) functionality on the Intel network card, the traffic will then be spread onto multiple receiver queues. The results are shown in Figure 6.

We notice that receive rate is lower than transmit rate, which is expected since packet reception involves more processing than packet transmission. Transmit rate reaches its maximum and saturates the link with two CPU cores, while receive rate is lower, although it increases when more CPU cores are added. In an attempt to further investigate receive rate, we perform a second measurement where we use all available CPU cores for receiving, and only vary the number of cores for transmission. We can observe that the results for receive rate is consistent with the previous configuration, and that receive rate is not limited by CPU capacity.

5.5. Latency

The purpose of this experiment is to investigate one-way latency. Latency can vary with traffic load, because of process scheduling and interrupt processing, for instance, so we want to measure latency while adjusting packet

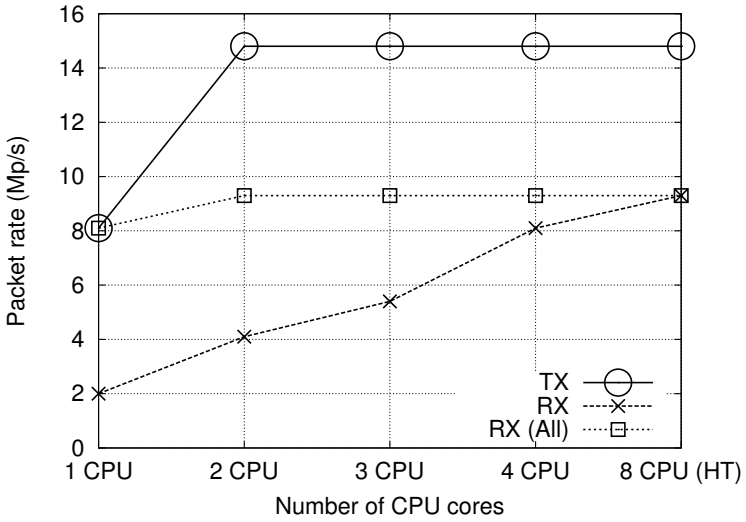

Figure 6: Maximum receive and transmit packet rate as a function of number of CPU cores. "TX" shows transmission rate as a function of the number of CPU cores used for transmission, while "RX" is the rate reported by the receiver. "RX (All)" shows the case where the maximum number of cores are used for reception, while varying the number of cores for transmission. In all cases, there is one receiver queue and one transmit queue per CPU core. The last measurement point "8 CPU (HT)" is with hyperthreading enabled. The packet rates are for one-minute transfers of UDP packets in 64-byte Ethernet frames, without user-specified rate. Packet bursts are enabled, with burst size 10.

rate. However, except for pktgen, none of the tools have the ability to measure latency with adjustable packet rate. Instead, for this experiment, we compare ping and pktgen. Ping measures two-way latency through ICMP Echo request/reply transactions between two machines, and permits to adjust the rate up to 1 kp/s.

One-way latency measurements depend on the generator and receiver clocks to be synchronized. Rather than having the generator and receiver on different machines and relying on clock synchronization, for instance through Precision Time Protocol (IEEE 1588)[28], we place the generator and the receiver on the same machine with a loopback cable between two interface ports.

In this experiment, we also want to evaluate the effects of power saving modes ("C-states") to assess the importance of proper experiment configuration. We run the measurements for C-states C0, C1, C3 and C6: C0 is without power saving; C1 halts the core when it is idle; C3 flushes caches,

stops core clocks, but leaves the voltage level unchanged; and C6 flushes caches, stops the core clocks, and lowers the voltage. Hence, C6 turns off most of the circuitery and therefore takes the longest time to wake up. Figure 7 shows one-way average latency for ping and pktgen, for C-states C0, C1, C3, and C6. Ping reports two-way latency, which we divide by two to get an estimate of one-way latency.

Pktgen with C-state C0 and C1 (there is no discernable difference between them) has the lowest latency, although there is a clear dependency on packet rate. It has slightly higher average latency at low rate (13.6 microseconds at 10 packets per second), where rate control is mainly through passive waiting, as described in 3.2. When the rate increases, there is more active waiting, and latency goes down. The lowest latency (9.1 microseconds) is at 10,000 packets per second. After that, latency goes up significantly as the rate increases. At high packet load, the NAPI interrupt moderation algorithm [29] attempts to reduce the amount of hardware interrupts and use polling mode instead, where the network interface periodically is checked for more incoming packets. This reduces overhead in the communication between network interface and driver, but at the expense of increased burstiness. Hence, average latency goes up.

The impact of power saving mode on latency is most noticeable at lower rates, where the CPU core is idle most of the time. The C-states with deeper sleep modes (C3 and C6) add substantial delay, compared to C0 and C1. As the rate increases, the CPU core gets more busy, and power saving has less effect.

Ping reaches its maximum rate with flooding mode, approximately 32 kp/s on our system, while pktgen has a maximum rate of approximately 840 kp/s. The main reason why ping flooding is so much slower is its stop-and-wait behavior, where it waits for a reply before sending the next request. Hence, ping's packet rate is limited by round-trip delay.

As expected, ping has slightly higher latency than pktgen, due to the fact that it is a user-space application and therefore has more overhead. Furthermore, ping exhibits considerably higher variations in latency. This can be seen in Figure 8, which shows the relative standard deviation for the measurements in Figure 7. The relative standard deviation for ping varies between 5% and 82%. In contrast, the relative standard deviation for pktgen is between 1% and 5% for all experiments. Since ping runs as a user process, it is exposed to user process scheduling and context switches, which cause larger variations. The last measurement point for ping is with flooding

enabled, where there is less impact from scheduling, and consequently the latency is lower.

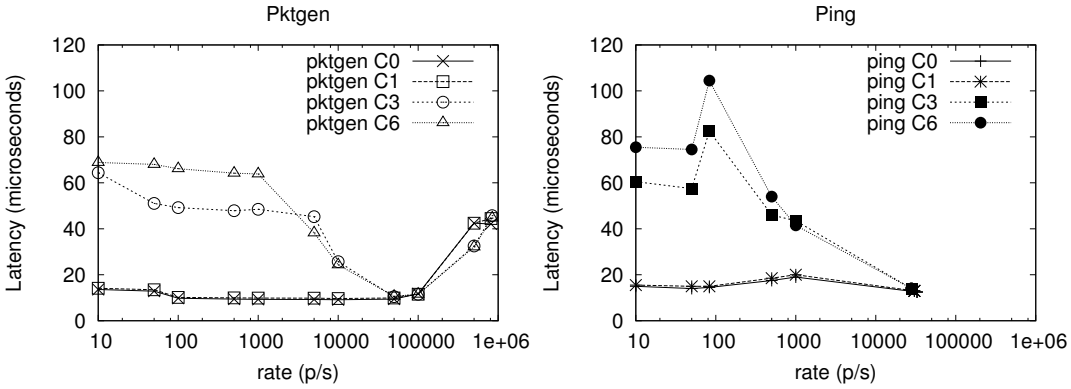

Figure 7: Average one-way latency. The graph shows average latency for one-minute transfers with UDP packets in 64-byte Ethernet frames and a single CPU core, for C-states C0, C1, C3 and C6. The last measurement point with ping (at approximately 32 kp/s) is with flooding enabled.

5.6. Packet delay variation

Packet delay variation, or jitter, is an important measure related to quality of service, and applications such as real-time audio and video can be sensitive to packet delay variation. Besides pktgen, it is only Iperf among the tools that measures packet delay variation. However, according to comments in the source code[3], it is not calculated correctly and provides only an average. Therefore, we only study packet delay variation for pktgen.

Figure 9 shows the packet delay variations from the latency measurements for pktgen in Section 5.5. Packet delay variation is computed in accordance with Section 2.3.1 as the difference in latency between two consecutive packets. The diagram in Figure 9 shows the average packet delay variation during the one-minute transfers. As expected, packet delay variations are significantly higher with C-states C5 and C6, since their sleep modes are deeper. For C0 and C1, packet delay variation is low, often below 1 microsecond, until the rate goes up and NAPI polling mode comes into operation, leading to more bursty arrivals and increasing variations in delay.

[3]Iperf 3.0.11, `https://github.com/esnet/iperf`. File src/iperf_udp.c, line 118.

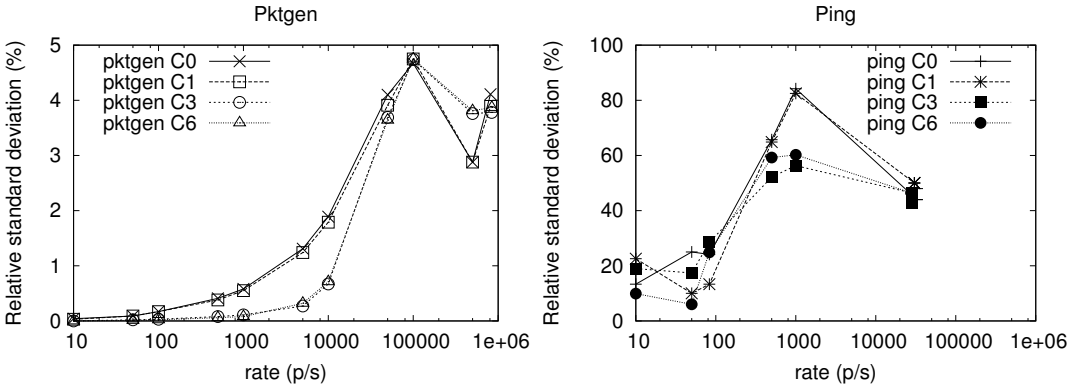

Figure 8: Relative standard deviation – the ratio of the standard deviation to the average – for one-way latency for one-minute transfers with UDP packets in 64-byte Ethernet frames and a single CPU core, for C-states C0, C1, C3 and C6. The last measurement point with ping (at approximately 32 kp/s) is with flooding enabled. Relative standard deviation for pktgen is much lower than for ping, so note the difference in scale on the y-axis for the two diagrams.

5.7. Discussion

The differences between the tools are mainly related to how they are designed. DPDK maps the complete network card memory to user space, thereby entirely bypassing the Linux driver. Netmap maps only the network card's ring buffers to user space and modifies the kernel drivers to support its API. Netmap's packet generator is a sample application tool that repetitively sends the same packet. Pktgen, in contrast, is a kernel process that uses standard Linux drivers, but bypasses the standard socket API for user space applications. Finally, Netperf and Iperf are user space tools that use the socket API, and consequently their performance is limited by operating system overhead.

In addition to being a tool for network performance analysis, pktgen is designed as a system tool for network device driver testing. Hence, a design goal for pktgen is that it should be a native Linux application and use the Linux kernel's data structures to manage packets. Those data structures are designed to suit many purposes and are relatively large. In contrast, Netmap and DPDK have their own structures, which are for a specific purpose and are kept at minimal size. This helps to keep down the amount of packet processing, but at the expense of less control.

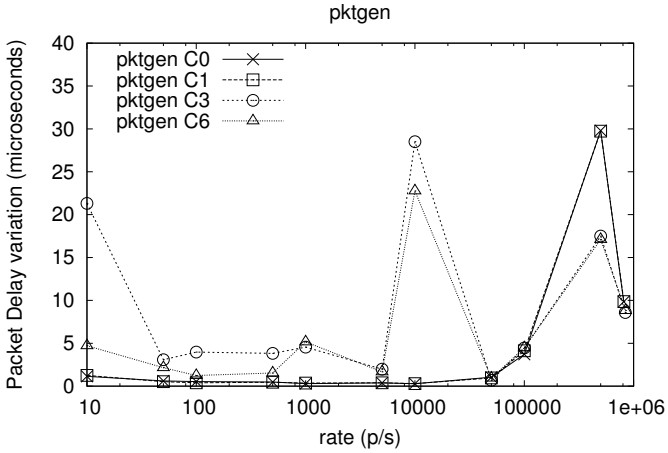

Figure 9: Packet delay variation for pktgen with C-states C0, C1, C3 and C6. The graph shows average packet delay variation for the same experiments as in Figure 7 and Figure 8, that is, one-minute transfers with UDP packets in 64-byte Ethernet frames and a single CPU core.

6. Conclusion

In this paper we present pktgen – a networking tool to generate and measure traffic at high speed, while still being fully integrated with Linux networking. We believe pktgen to be useful for a wide range of applications: for instance, development of drivers for network cards and other system-level software; analysis and improvements of open source routers and other networking equipment; performance benchmarking of routers and switches; troubleshooting networks by studying their behavior under varying traffic conditions; network security analysis by investigating resistance to denial-of-service attacks, and so on.

Pktgen is designed to exploit new features available in modern computer systems, such as multicore processors and network cards with support for multiple queues. Since it runs in the kernel, it has little overhead and can get the highest performance out of the system.

We have studied performance tests with pktgen and made comparisons with other tools, including DPDK, Iperf, Netmap, and Netperf. Iperf and Netperf run in user-space and exhibit significantly lower throughput than the other tools, due to the overhead in the socket API. DPDK, Netmap, and pktgen circumvent the socket API in different ways. DPDK and Netmap run

in user-space but have their own custom-made libraries for accessing network interface hardware. In contrast, pktgen runs in the Linux kernel, with close access to the hardware, and has a generic design so that it can run on all network devices for which there are Linux drivers.

In addition to throughput measurements, we conduct experiments with latency measurements. Except for pktgen, none of the investigated tools support latency measurements with rate control. Instead, for the sake of comparison, we use "ping". The purpose of the latency measurements is to establish a baseline for a wider range of measurements, and the results indicate that pktgen can be used reliably for latency measurements with variations less than a few microseconds. We note though that in order to avoid artifacts from operating system scheduling, a latency measurement needs to be performed within a certain range of packet rates, which serves to illustrate the importance of careful calibration of the measurement tool.

Pktgen with all features described here is available at `https://people.kth.se/~danieltt/pktgen`. Moreover, pktgen's generator module is available as a part of the standard Linux kernel distribution.

References

[1] R. Jones, Netperf: a network performance benchmark [cited 30-sep-2015].
URL `http://www.netperf.org/netperf/`

[2] Iperf. TCP and UDP bandwidth performance measurement tool [cited 30-sep-2015].
URL `https://code.google.com/p/iperf/`

[3] R. Olsson, pktgen the linux packet generator, in: Linux Symposium 2005, Ottawa, Canada, 2005, pp. 11–24.

[4] IA-PC HPET (High Precision Event Timers) Specification, Tech. rep., Intel (2004) [cited 30-sep-2015].
URL `http://www.intel.es/content/dam/www/public/us/en/documents/technical-specifications/software-developers-hpet-spec-1-0a.pdf`

[5] Z. Yi, P. P. Waskiewicz, Enabling Linux Network Support of Hardware Multiqueue Devices, in: Linux Symposium, Ottawa, Canada, 2007, pp. 305–310.

[6] R. Love, Kernel korner: CPU affinity, Linux J. 2003 (111) (2003) 8– [cited 30-sep-2015].
URL http://www.linuxjournal.com/article/6799

[7] O. Hagsand, R. Olsson, J. Låås, B. Gördén, Measurement of IP forwarding performance on complex computer architectures, in: Swedish National Computer Networking Workshop 2011.
URL http://www.herjulf.se/robert/papers/sncnw_2011.pdf

[8] S. Moreaud, B. Goglin, Impact of numa effects on high-speed networking with multi-opteron machines, in: Proceedings of the 19th IASTED International Conference on Parallel and Distributed Computing and Systems, PDCS '07, ACTA Press, Anaheim, CA, USA, 2007, pp. 24–29.
URL http://hal.inria.fr/docs/00/17/57/47/PDF/PDCS07.pdf

[9] J. R. Jesper Dangaard Brouer, John Fastabend, Tx bulking and qdisc layer, in: Linux Plumbers Conference, 2014 [cited 30-sep-2015].
URL http://people.netfilter.org/hawk/presentations/LinuxPlumbers2014/performance_tx_qdisc_bulk_LPC2014.pdf

[10] Ostinato [cited 30-sep-2015].
URL http://ostinato.org

[11] Multi-Generator (MGEN) [cited 30-sep-2015].
URL http://www.nrl.navy.mil/itd/ncs/products/mgen

[12] (C)RUDE [cited 30-sep-2015].
URL http://rude.sourceforge.net/

[13] A. Botta, A. Dainotti, A. Pescapé, A Tool for the Generation of Realistic Network Workload for Emerging Networking Scenarios, Computer Networks 56 (15) (2012) 3531–3547. doi:10.1016/j.comnet.2012.02.019.

[14] Linux Manual TTCP [cited 30-sep-2015].
URL http://linux.die.net/man/1/ttcp

[15] L. Rizzo, Revisiting network I/O APIs: the netmap framework, Communications of the ACM 55 (3) (2012) 45–51. doi:10.1145/2093548.2093565.

[16] Intel, Intel DPDK: data plane development kit [cited 30-sep-2015].
URL http://dpdk.org/

[17] N. Bonelli, A. Di Pietro, S. Giordano, G. Procissi, Flexible high performance traffic generation on commodity multi—core platforms, in: Proceedings of the 4th International Conference on Traffic Monitoring and Analysis, TMA'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 157–170. doi:10.1007/978-3-642-28534-9_17.

[18] N. Bonelli, S. Giordano, G. Procissi, R. Secchi, Brute: A high performance and extensible traffic generator, in: Proc. of SPECTS, 2005, pp. 839–845.

[19] S. Zander, et al., KUTE – A High Performance Kernel-based UDP Traffic Engine, Tech. rep., Melbourne, Australia (2005) [cited 30-sep-2015].
URL http://caia.swin.edu.au/genius/tools/kute/

[20] F. Schneider, IPerformance Evaluation of Packet Capturing Systems for High-Speed Networks, Master's thesis, Technische Universitat Munchen (2005) [cited 30-sep-2015].
URL http://www.net.t-labs.tu-berlin.de/~fabian/papers/da.pdf

[21] IXIA, IxNetwork Datasheet [cited 30-sep-2015].
URL http://www.ixiacom.com/products/ixnetwork

[22] Spirent, Smartbits [cited 30-sep-2015].
URL http://www.spirent.com/Products/Smartbits.aspx

[23] International vocabulary of metrology – basic and general concepts and associated terms (2012) [cited 21-sept-2015].
URL http://www.bipm.org/utils/common/documents/jcgm/JCGM_200_2008.pdf

[24] A. Botta, A. Dainotti, A. Pescape, Do you trust your software-based traffic generator?, Communications Magazine, IEEE 48 (9) (2010) 158–165. doi:10.1109/MCOM.2010.5560600.

[25] S. Hayes, Controlling Processor C-State Usage in Linux, Tech. rep., Dell (2013) [cited 30-sep-2015].
URL http://en.community.dell.com/cfs-file.ashx/__key/telligent-evolution-components-attachments/13-4491-00-00-20-22-77-64/Controlling_5F00_Processor_5F00_

C_2D00_State_5F00_Usage_5F00_in_5F00_Linux_5F00_v1.1_5F00_Nov2013.pdf

[26] Interrupt Moderation Using Intel Gigabit Ethernet Controllers Application Note (AP-450), Tech. rep., Intel (2007) [cited 30-sep-2015].
URL http://www.intel.com/content/dam/doc/application-note/gbe-controllers-interrupt-moderation-appl-note.pdf

[27] perf: Linux profiling with performance counters [cited 30-sep-2015].
URL https://perf.wiki.kernel.org

[28] K. Lee, J. Eidson, IEEE-1588 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems, in: In 34 th Annual Precise Time and Time Interval (PTTI) Meeting, 2002, pp. 98–105.

[29] J. H. Salim, R. Olsson, A. Kuznetsov, Beyond softnet, in: 5th Annual Linux Showcase & Conference (ALS '01), 2001, pp. 165–172 [cited 25-sept-2015].
URL https://www.usenix.org/publications/library/proceedings/als01/full_papers/jamal/jamal.pdf