Mind the Gap – A Comparison of Software Packet Generators

Paul Emmerich
Technical University of Munich
Garching, Germany
emmericp@net.in.tum.de

Sebastian Gallenmüller
Technical University of Munich
Garching, Germany
gallenmu@net.in.tum.de

Gianni Antichi
University of Cambridge
Cambridge, United Kingdom
gianni.antichi@cl.cam.ac.uk

Andrew W. Moore
University of Cambridge
Cambridge, United Kingdom
andrew.moore@cl.cam.ac.uk

Georg Carle
Technical University of Munich
Garching, Germany
carle@net.in.tum.de

ABSTRACT

Network research relies on packet generators to assess performance and correctness of new ideas. Software-based generators in particular are widely used by academic researchers because of their flexibility, affordability, and open-source nature. The rise of new frameworks for fast IO on commodity hardware is making them even more attractive. Longstanding performance differences of software generation versus hardware in terms of throughput are no longer as big of a concern as they used to be few years ago.

This paper investigates the properties of several high-performance software packet generators and the implications on their precision when a given traffic pattern needs to be generated. We believe that the evaluation strategy presented in this paper helps understanding the actual limitations in high-performance software packet generation, thus helping the research community to build better tools.

CCS Concepts

•**Hardware** → *Networking hardware;* •**Networks** → **Network experimentation; Network performance analysis;** *Network performance modeling; Network measurement; Packet-switching networks;*

Keywords

NetFPGA; Packet Processing Frameworks; Benchmarking

1. INTRODUCTION

Computer networks are one of the greatest accomplishments of the 21st century. Our society has become dependent on the permanent availability and security of its ICT infrastructure. This distinguished position subjects it to constant development and research. Researchers rely on packet generators to test their ideas and to assess the correct operation of their proof-of-concepts. This places packet generators at the forefront of research as innovation enablers. Validating a new apparatus under different loads and understanding its performance in terms of throughput or latency is the first step of the production life-cycle. Moreover, the ability to generate synthetic traffic makes packet generators a key instrument for field engineers troubleshooting networks.

Packet generators are implemented over both hardware and software platforms. The former are typically closed-source and expensive, but more accurate. The latter are cheaper, slower, less accurate, but most importantly usually open-source. As the open-source nature allows for easy modifications and extensions of core features, software packet generators are very popular in academic research. The rise of new fast IO frameworks, such as netmap or DPDK [2, 25] is making the software generation even more attractive as it can cope with high packet rates on commodity NICs.

This paper investigates the properties of several high-performance software packet generators based on the aforementioned fast IO frameworks. We conduct an analysis to study the implications which might rise when a given traffic pattern needs to be reliably generated at high rates. This is a core aspect in the evaluation of a packet generator because the generated pattern can influence the performance of a system under test [18]. RFC 1242, a collection of terminology for benchmarking network devices, defines constant load as "fixed length frames at a fixed interval time," i.e., constant bit-rate (CBR) traffic [15]. This definition is often used, e.g., by the RFC 2544 benchmarking standard or modern benchmarking efforts like the VSWITCHPERF project [28].

Hence, we focus most of the analysis on comparing software packet generators against CBR traffic. Further, we discuss the impact of a non-CBR traffic pattern and different CPU architectures on the performance of packet generators. In particular, we choose the Poisson process as network traffic assumes a self-similar distribution and traffic generated by a Poisson process can approximate real traffic patterns for short time spans [26]. Finally, we present a study on the impact of fast IO frameworks over software timestamping accuracy when a packet generator needs to be used for latency measurements. By presenting our study we highlight the limitations and misconceptions associated with today's packet generators fostering the development of better tools for our research community. Our main contributions are:

- A comparative analysis of a number of software packet generators when a CBR traffic pattern and a non-CBR one need to be generated.
- The influence of different CPU microarchitectures on the traffic generation process.
- The impact of fast IO frameworks on software timestamping accuracy when a packet generator needs to be used for latency measurements.

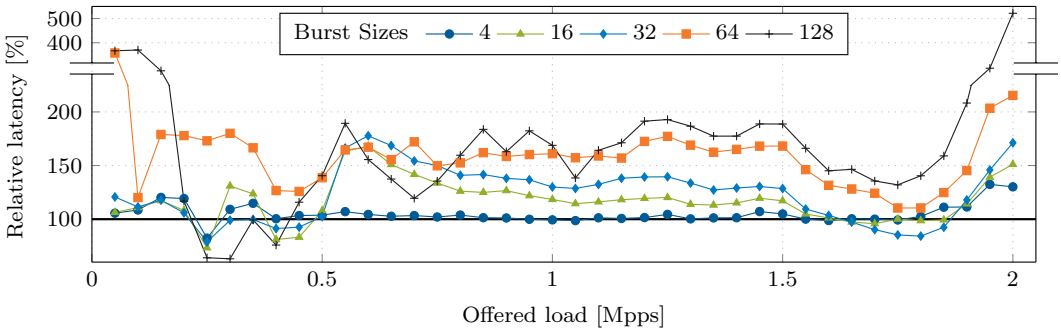


Figure 1: Relative observed latency of different burst sizes

The rest of the paper is organized as follows: Section 2 proposes a motivating example highlighting the effect of different packet generation schemes on experimental results. Section 3 and Section 4 describe the packet generators being analyzed in this paper and our measurement setup. We investigate the precision and accuracy of different rate control mechanics in Section 5, while we study the influence of different CPU architectures on the packet generation process in Section 6. Latency measurement capabilities are discussed and evaluated in Section 7. Finally, we review related work in Section 8 and conclude our paper in Section 9.

2. IMPACT OF PACKET GENERATION: A MOTIVATING EXAMPLE

The behavior of a system under test changes measurably when the traffic pattern on its ingress interfaces is altered [18]. RFC 2544 [16] defines a number of tests that may be used to describe the performance characteristics of a network interconnecting device. It specifies, in particular, CBR as required input traffic pattern. It is thus important that a packet generator can generate CBR reliably.

State-of-the-art software packet generators rely on processing packets in large *batches* in their default configuration. Batching is a common technique to increase the performance of packet processing frameworks (see Section 5.2). These batches lead to *bursts* on the wire, resulting in traffic patterns significantly different from the expected CBR traffic[1].

To this end we investigate the impact of the burst size on the forwarding latency of a device under test: Open vSwitch 2.0.0 on Debian running a Linux kernel v3.7.9 restricted to a single 3.3 GHz CPU core. We measure the forwarding latency of this system under increasing load in steps of 0.05 Mpps up to its maximum capacity of 2 Mpps at 64 byte packets. We first run the experiment with a CBR traffic pattern, and then repeat the measurement with increasing explicitly configured burst sizes. These burst sizes effectively emulate a packet generator with a larger batch size. We use the MoonGen packet generator with the CRC rate control method [18] to precisely generate the burst sizes. Latency measurements are performed using MoonGen's support for hardware capabilities (Intel 82599 NIC), allowing sub-microsecond measurements [18]. Figure 1 plots the median latency against the generated packet rates and burst sizes. The latency is expressed as relative with respect to the CBR case. Increasing burst size increases the observed latency. Bursts of 4 packets have already a large impact: the worst-case deviation is 32%. This figure demonstrates that a system under test can report different behavior depending on the input traffic pattern. As the aforementioned RFC argues the need for CBR traffic, assessing the reliability of the generator itself first is important. Moreover, analytically removing the additional latency introduced by the burst is not possible. Given that transmitting each packet on a 10 Gbit/s link takes 67.2 ns, a burst size of 128 packets corresponds to 8.6 μs time on the wire. The absolute difference in latency between CBR and bursts of 128 packets is between 50 μs and 100 μs and varies with the packet rate. This effect is visible for other burst sizes too, demonstrating the impossibility to treat the effects of bursts as manageable measurement artifacts.

Growing internal batch sizes lead to higher performance for packet generators. However, bursty traffic by definition does not generate a constant bit rate. If we want to obtain meaningful results (RFC 2544 compliant), it is important not to compromise the precision of CBR generation to achieve greater overall throughput. We notice that by default, most of the software packet generators use batch sizes between 16 and 512 packets resulting in bursts of equivalent sizes. Therefore, it is crucial to understand the limitations of the software packet generators being used for testing and the trade-off between their precision and performance.

3. SOFTWARE PACKET GENERATORS IN 2016

Software packet generators can be distinguished in two different classes depending on the generation mechanism.

Traditional software packet generators rely on the interface provided by the kernel for packet IO. Using a standard OS interface enables a high degree of compatibility and flexibility. In fact, the wide range of features supported by the network stack of an OS can be used by these packet generators. Therefore, the user does not have to re-implement protocols already supported by the OS itself. The main drawbacks are related to precision and performance. The former is due to the employed timing functions [23], while the latter is caused by the network stack itself which is optimized for compatibility and stability rather than high performance or high precision [12, 25].

[1]We refer to internal processing of multiple packets together as *batches* and to the resulting back-to-back frames on the wire as *bursts* throughout this paper.

This paper investigates how packet generators designed for 1 Gbit platforms (traditional ones) differ from packet generators designed for 10 Gbit NICs. As these traditional packet generators are still used, we want to investigate how they perform on today's 10 Gbit platforms. D-ITG [1] and trafgen [9] are two examples, optimized for different design goals but using the same standard Linux IO APIs. We choose trafgen as traffic generator specialized in high-performance packet generation (in the context of 1 Gbit networks), while D-ITG because of its ability to generate both realistic and synthetic traffic patterns [14] with high precision. Further, it supports distributions such as normal, Pareto, Cauchy, gamma, and Weibull beside CBR.

Modern software packet generators use special frameworks which bypass the entire network stack of an OS. They are optimized for high speed and low latency at the expense of compatibility and support for high-level features. The user in this case has to re-implement protocols on top of these frameworks. Those architectural changes overcome the main drawbacks of traditional packet generators. They reduce the number of costly context switches or avoid them entirely [19] and rely on polling and busy waiting for precise timing, eliminating the main cause of packet transmission inaccuracy identified by Botta et al. and Paredes-Farrera et al. [13,23]. The dependency of the packet generation process on CPU load can be avoided by using dedicated cores for packet generation on modern multicore CPUs. The main drawback of these frameworks is the dependency on specialized drivers, creating hardware dependency and limited compatibility.

As the main focus of the paper is a comparative analysis of high-performance packet generators, we included, to the best of our knowledge, as many as available to cover a wide range of high-performance frameworks. Both Windriver System's Pktgen-DPDK [8] and MoonGen [17] are based on the DPDK framework. We deliberately choose not the most recent version of Pktgen-DPDK but rather a version which uses the same version of DPDK as MoonGen to make a fair comparison: later versions of DPDK are slightly slower in our tests due to increased overhead. The code affecting precision and timing in Pktgen-DPDK was not modified since the version we use here.

Pkt-gen [4] is a widely used packet generator included in netmap while PFQ offers pfq-gen [6]. PF_RING ZC [7], a high-performance framework found in production systems provided by ntop [5], offers the packet generator zsend. Table 1 summarizes the investigated software packet generators.

	Version	IO API
D-ITG [1]	v2.8.1	Linux
trafgen [9]	v0.5.7	Linux
Pktgen-DPDK [8]	v2.8.0	DPDK (v1.8.0)
MoonGen [17]	git 5cf96c72*	DPDK (v1.8.0)
	git bfe8b5b1†	
	git 39e0cb64‡	
pkt-gen [4]	git b24fce99	netmap
pfq-gen [6]	v5.2.9	PFQ
zsend [7]	v6.3.0.160209	PF_RING ZC

*) Used for CBR traffic and hardware timestamping
†) Used for Poisson traffic
‡) Used for software timestamping

Table 1: Investigated software packet generators

4. TEST SETUP

Our main test setup consists of two machines, directly connected via a 10 Gbit/s fibre link. One is equipped with a NetFPGA-10G [3] programmed with OSNT [10] for high precision packet inter-arrival time characterization. The other, which is used to run the software packet generators, has an Intel i7-960 CPU with a base frequency 3.2 GHz and an Intel X520 NIC (based on the Intel 82599 Ethernet controller). Ubuntu Linux 14.04 LTS (kernel 3.16) is the chosen OS.

4.1 OSNT

OSNT is a fully open-source hardware traffic generation and capturing system. Its architecture is motivated by limitations in existing hardware network testing solutions: proprietary/closed-source, high costs, and inflexibility. Primarily designed for the research and teaching community, its key design goals are low cost, high-precision time-stamping, packet transmission, and scalability. In addition, the open-source nature of the system allows extensibility and adding new protocol tests to the system. The prototype implementation builds upon the NetFPGA-10G platform – an open-source hardware platform designed to support full line-rate programmable packet processing. The combination of traffic generator and traffic monitor subsystems into a single FPGA-equipped device allows a per-flow characterization of a networking system within a single card. This paper concentrates on the use of OSNT for its monitoring capabilities rather than packet generation. In particular, the OSNT traffic capture subsystem is used to provide high-precision inbound timestamping.

Timestamping mechanism

OSNT provides an accurate timestamp mechanic for incoming packets. Packets are timestamped as close to the physical Ethernet device as possible to minimize jitter and permit accurate latency measurement. A dedicated timestamping unit stamps packets as they arrive from the physical (MAC) interfaces with a practical resolution of 6.25 ns. OSNT uses Direct Digital Synthesis (DDS), a technique by which arbitrary variable-frequencies are generated using synchronous digital logic [27] to correct the frequency drift of the FPGA oscillator. The addition of a stable pulse-per-second (PPS) signal such as that derived from a GPS receiver permits both high long-term accuracy and the synchronization of multiple OSNT elements.

5. PACKET GENERATION: QUALITATIVE AND QUANTITATIVE ANALYSIS

Rate control is the mechanism implemented by a traffic generator to assure the generated traffic matches the required characteristics. The proposed analysis focuses on assessing the rate control capabilities of the investigated packet generators based on:

- **Bandwidth:** refers to the maximum throughput can be obtained from the generation process (how fast is it in terms of packets per second?).

- **Accuracy:** describes the systematic errors, a measure of statistical bias (how close is the average observed rate to the configured one?).
- **Precision:** describes the random errors, a measure of statistical variability (how much do individual inter-packet gaps deviate from the configured value?).

We use the term accuracy to estimate how close the average of a set of measurements matches the target. Precision refers to the deviation of an individual measurement, such as the inter-arrival time between two packets, from the target. For instance, a packet generator configured to produce constant inter-arrival times which generates bursty traffic would be classified as accurate if the overall average rate is correct. The precision in contrast would be low due to differences of inter-burst and intra-burst packet gaps.

5.1 Rate control: three different approaches

Rate control is a feature available in hardware on commodity NICs such as the Intel 82599 [20] or Intel XL710 [22]. Software packet generators can use this property to increase their precision, which we define as **hardware supported approach**.

The **pure software approach** simply waits for a configured time between sending individual packets. This entails precision problems: sleep functions provided by the OS are not reliable as their granularity is limited [13, 23]. Busy waiting techniques can solve this. However, abstractions from the OS and driver can lead to unintended buffering and high costs for the required system calls to send individual packets. Modern packet generators solve this issue with specialized IO frameworks that provide full access to the hardware. Unfortunately, drivers cannot send packet data directly to a NIC: they can only place it in a DMA memory region and inform the NIC to fetch it asynchronously. This causes unwanted jitter in the required transmission time due to the two required PCIe round trips, DMA coalescing on the NIC, and potential buffering on the NIC – this jitter cannot be removed by a pure software solution.

An alternative method of pure software rate control injects *null* packets between real ones to adjust the desired inter-packet gap [18]. For example, instead of waiting between the generation of two packets, MoonGen sends invalid packets by corrupting the CRC checksum. The inter-packet gap is determined by the length of invalid packets in between two valid packets. Tests show that it is also possible to send invalid packets violating the 60 byte minimum for even shorter gaps. This mechanism proves to be reliable if the device under test is capable of dropping the invalid packets efficiently in hardware without affecting the tested processing steps, which is the case for most devices. We refer to this as **corrupted CRC approach**.

5.2 Performance vs. precision

The most common practice to reach high IO throughput is sending large batches of packets to the driver instead of individual packets [19,25]. This leads to a trade-off between speed and precision for packet generators when the user does not require bursty distributions. Indeed, all the aforementioned frameworks for high-speed packet IO use large batch sizes by default: Table 2 shows the defaults for the investigated packet generators. This implementation choice is suitable for general purpose packet processing applications, but not for precise packet generation. However, setting the batch size to one packet allows estimating the precision of a traffic generator in its best case scenario.

Packet generator	Default Batch size	Throughput (Default) [Mpps]	Throughput (Precise) [Mpps]
MoonGen (HW)	63	14.88	13.52[1]
MoonGen (CRC)	N/A	N/A[2]	5.74
MoonGen (SW)	1	N/A[2]	5.36
zsend	16	14.84	14.71[3]
Pktgen-DPDK	16	14.88	4.54
pfq-gen	32	5.67	3.59
netmap pkt-gen	512	14.88	1.55
D-ITG	1	N/A[2]	0.22
trafgen	?[4]	0.40	N/A[4]

[1]) Intel 82599, highest reliable hardware setting
[2]) No imprecise generation possible
[3]) Not precise at high rates despite configuration
[4]) Batch size unclear, failed to hit target rate within ±10%

Table 2: Achieved throughput on a Core i7-960

We first run performance tests using the default settings for each packet generator. Then, we configure each of them to be as precise as possible (forcing the batch size to one packet) and determine the maximum rate by increasing the packet rate setting in steps of 1 Mpps (0.05 Mpps for packet generators that do not reach 1 Mpps). These tests allow us to assess the impact of precise configuration over the performance. Table 2 reports the obtained results. MoonGen (HW) refers to the MoonGen version with hardware support (Intel 82599) for the rate control, while MoonGen (CRC) enables the rate control with the corrupted CRC approach. Finally, MoonGen (SW) has a pure software implementation of the rate control.

The impact of different CPU microarchitectures is evaluated in Section 6. The low performance experienced with netmap pkt-gen in its precise settings is due to netmap's architecture itself. netmap relies on system calls for packet IO and uses a burst size of 512 in its default setting to compensate for its costly transmit operation. While this solution is beneficial for security and stability [25], it results in poor performance when a precise packet generation time is required. In fact, reducing the batch size to one packet causes a system call for every packet, affecting the overall throughput.

Both trafgen and D-ITG do not rely on fast IO frameworks. Indeed, they were architected for 1 Gbit links. D-ITG can provide per-packet statistics. However, it reported incorrect results at rates above 0.1 Mpps.

5.3 Accuracy

Accurately hitting the target rate is important for the reproducibility of experiments. Most of the packet generators reliably generate the requested packet rates (unless overloaded) within a relative error of less than 0.2%. Table 3 shows packet generators that fail to do so. Trafgen does not claim to be accurate: the rate control setting is called "Interpacket gap in us (approx)". The versions of MoonGen which rely on hardware features on commodity Intel

Packet generator	Target [Mpps]	Measured [Mpps]	Rel. error
trafgen	0.1	0.069	31%
	0.01	0.006	36%
MoonGen (82599 HW)	1	1.00	< 0.1%
	4	4.00	< 0.1%
	8	7.98	0.28%
MoonGen (XL710 HW)	1	1.03	3.3%
	4	4.08	2%
	8	8.17	2%
Pktgen-DPDK	0.976	0.840	16%
	2.54	2.17	17%
	4.1	3.45	19%

Table 3: Accuracy evaluation

NICs fail the test as well in some cases. The hardware rate limiting features of these NICs are not designed for precise packet generation but rather for limiting applications where a coarse approximation is sufficient and short bursts may even be desirable.

Pktgen-DPDK provides a granularity of 0.195 Mpps, leading to the odd target values shown in Table 3. In addition, the rate control algorithm of Pktgen-DPDK is incorrect: it assumes that generating and sending a packet does not take a significant time and waits for a fixed time between sending two packets. This assumption is not valid, leading to the poor accuracy.

5.4 Precision

We target small inter-frame gaps (high load on the system) to evaluate their precision under stress condition. In fact, the higher the packet rate, the lower the requested inter-frame gap, leading to higher requirements in terms of precision as the generated traffic will likely be characterized by micro-bursts (back-to-back frames). Increasing the rate impacts also the system itself, potentially decreasing the precision, thus amplifying the problem.

As most of the packet generators fail to achieve high packet rates (cf. Section 5.2), we use a packet size of 128 bytes (including CRC) for the evaluation. Packet size typically does not influence the performance of packet generation [25]. This setting allows us to reduce the inter-frame gap and achieve relatively high bandwidths.

CBR traffic is the hardest pattern to generate precisely as each gap must have exactly the same length, i.e., the resulting histogram should ideally consist of just a single bucket. It also allows for an easy visual comparison of the precision as well as an analytic quantification by determining the mean squared error (MSE) in nanoseconds^2.

5.4.1 *Rate control: software implementations*

Differences in precision between software packet generators stem from different rate control implementations. Short time intervals are hard to measure due to the granularity of underlying timers. x86 CPUs feature the RDTSC instruction which returns a cycle count of the CPU, enabling a cycle-level granularity. This cycle counter is independent of the actual frequency due to power-saving or Turbo Boost and synchronized across CPUs on all modern CPUs. System

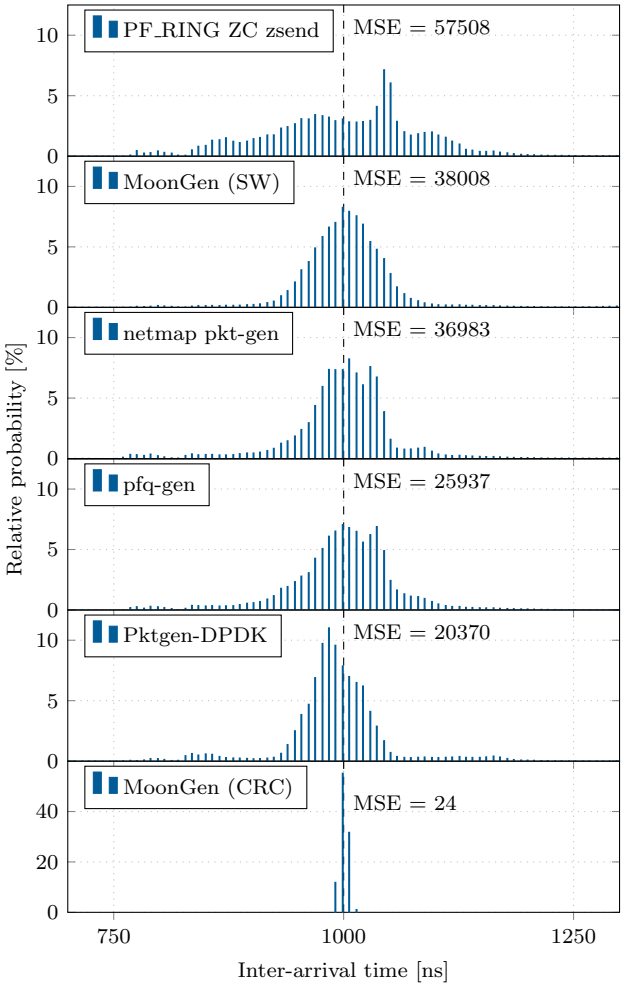


Figure 2: Software rate control at 1 Mpps

calls can use timers with coarser granularity, which may not be appropriate for nanosecond time spans that high-speed packet generators need to deal with. In the following we propose a brief description of the method being adopted by each packet generator:

PF_RING ZC zsend uses a separate thread with the purpose of calling `clock_gettime()` (with parameters that map to RDTSC on the system) and storing the result in a memory location in a tight loop to alleviate the overhead to the system call. The transmit thread then uses another busy-wait loop until the counter reaches the transmit time for a packet before sending packets to the driver. Neither thread uses memory fences. The transmit thread is therefore not guaranteed to see the most recent store by the timestamping thread.

Pktgen-DPDK uses RDTSC directly in a busy-wait loop for a fixed time between passing packets to the driver. This leads to a poor accuracy as explained in Section 5.3. For better comparison of the precision in the following tests, we opted to empirically determine the packet rate setting, and hence the fixed wait time, such that its self-reported transmit rate matches our target rate as closely as possible.

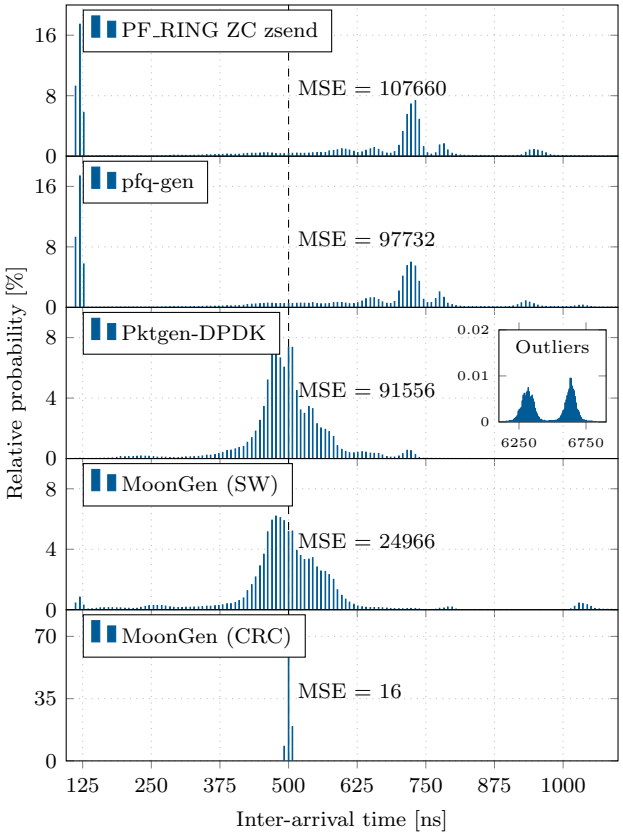


Figure 3: Software rate control at 2 Mpps

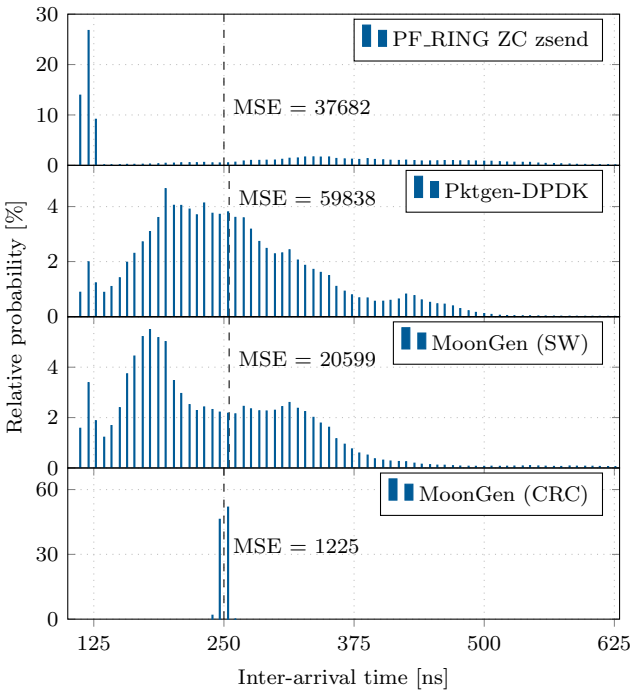


Figure 4: Software rate control at 4 Mpps

PFQ pfq-gen consists of three parts: the userspace application, the PFQ kernel module and the NIC driver. pfq-gen stores the desired transmit time in the packet metadata which is evaluated by the PFQ kernel module (running on a separate core). The kernel module waits for the specified time by calling the Linux kernel function `ktime_get_real()` in a busy-wait loop. Note that this function does not use the RDTSC instruction to determine time on each call, limiting the precision.

netmap pkt-gen uses the `clock_gettime()` system call (with parameters that map to RDTSC on the system) in a busy-wait loop before passing packets to the driver via a system call.

MoonGen pure software rate control adopts a solution similar to PFQ pfq-gen. It embeds the desired inter-packet gap in the packet metadata and send the packets via a lock-free queue to a transmit thread running C code (opposed to Lua in the main thread). The transmit thread uses a busy-wait loop employing RDTSC to achieve the highest possible precision.

5.4.2 Results at 1 Mpps

Figure 2 shows the measured inter-arrival time at 1 Mpps. We judge the precision by the shape of the distribution and the MSE value. High precision is expressed by a narrow distribution accompanied by a low MSE. PF_RING ZC's timestamping thread suffers from the lack of memory fences on a multi-core CPU: the transmit task does not see the latest value. This effectively limits the timer's granularity as the value is not updated as often as expected, explaining the large deviation present on its distribution resluting in a high MSE. The deviation of the CRC-based approach is within the precision of NetFPGA's timestamping mechanic and shows a narrow distribution, ergo a low MSE.

5.4.3 Results at higher rates

Figures 3 and 4 show the histograms at 2 Mpps and 4 Mpps respectively for packet generators that are still able to cope with these rates in the precise settings. Both zsend and pfq-gen start to generate bursts. They follow a very similar distribution at 2 Mpps, indicating that the root cause of the bursty traffic is the same. The only component shared by these two traffic generators is the Intel ixgbe kernel driver in a slightly modified version. This explains how zsend was able to generate 14.7 Mpps in the performance test even in the precise settings: it is actually not precise at higher rates. We tried to use DPDK to send packets without batching and found a hardware limit of 12.9 Mpps (using 3 cores/queues, adding another did not increase the performance) when not using batches. This demonstrates that there must be unintentional batching of packets to achieve high performance. Pktgen-DPDK and MoonGen are not affected by this problem as both of them use the DPDK userspace variant of the ixgbe driver which employs a completely different transmit path.

Pktgen-DPDK exhibits another issue at higher rates: there are additional peaks in the distribution between 6 000 ns and 7 000 ns. This leads to the MSE values that are far worse than expected from a visual inspection of the histograms. The problem is not caused by DPDK as MoonGen is not

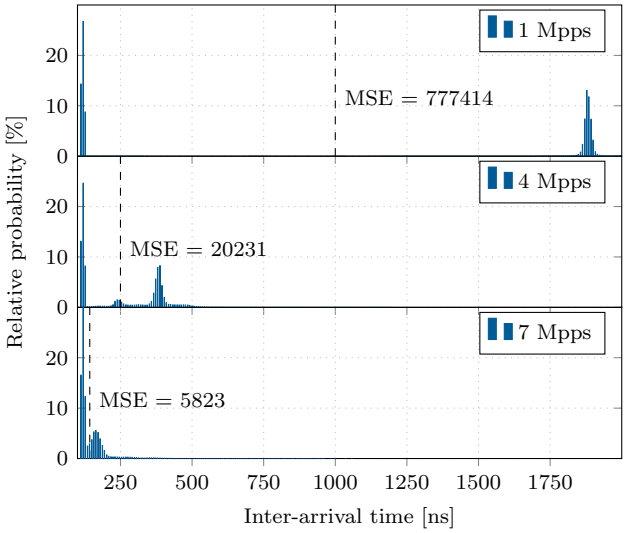


Figure 5: MoonGen with Intel 82599 hardware rate control

affected by this. The most likely reason is that the transmit thread in Pktgen-DPDK calculates and reports statistics regularly while in MoonGen this is done in a separate thread and via hardware counters. The CRC-based approach also shows a higher MSE than expected: there are a total of 10 outliers between 3 μs and 46 μs every 128 packets in the first 1 280 generated packets. This is caused by the insufficient performance of the Lua code at higher rates before the initial just-in-time compilation.

5.4.4 Hardware support on commodity NICs

The test campaign conducted so far studies the impact of different software implementation for the rate control. This section investigates the benefits that the hardware support can bring for precision and accuracy in packet generation. Hardware rate control features are available on NICs based on Intel 82599, X540, and XL710 network chips. Tests show that the Intel X540 10GBASE-T chip is more precise than software implementations on DPDK and PF_RING ZC [18]. However, this evaluation was restricted to 1 Gbit/s due to restrictions of the measurement setup. Here, we evaluate NICs based on Intel 82599 and XL710 chips which feature SFP+ ports (10 Gbit/s). The software packet generator of choice is MoonGen as it ships with the feature to enable hardware rate control on both cards. After activating, the NIC handles the entire rate control process.

The Intel 82599 datasheet outlines the rate control algorithm used in Section 7.7.2.1 [20]. Figure 5 shows that the hardware generates bursts of two packets at rates of 1, 4, and 7 Mpps on Intel 82599 NICs. The algorithm described in the datasheet does not match this observation, highlighting problems when relying on black-box hardware for reproducible experiments. The newer XL710 rate control lacks a detailed explanation of its inner working. Our measurements show that it generates bursts between 64 and 128 packets depending on the rate and packet size. Figure 6 shows the upper part of a CDF as these large bursts are not easily visualized in a histogram.

5.4.5 Precision at low speeds

D-ITG and trafgen are both too slow for 10 Gbit/s connections, so we handle these two separately with packet rates of 0.01 and 0.1 Mpps. Figure 7 and 8 show histograms of the measured inter-arrival times of trafgen and D-ITG respectively. Trafgen is not only inaccurate (note that the target line does not even appear on the figure) and slow, but also imprecise.

D-ITG generates a bimodal distribution oscillating around the target rate. The two modes of the distribution are only $\approx$ 200 ns away from the target at both tested rates. It is, however, prone to bursts as the rate increases. 0.5% of the

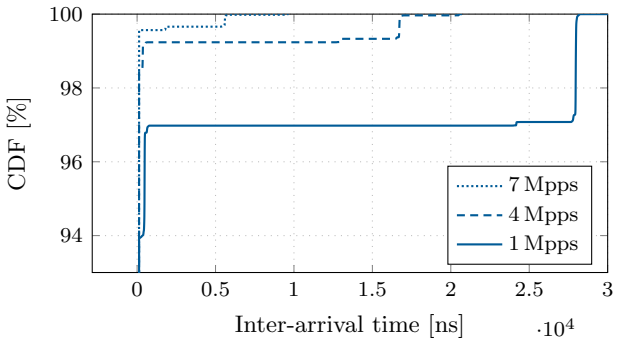


Figure 6: MoonGen with Intel X710 hardware rate control

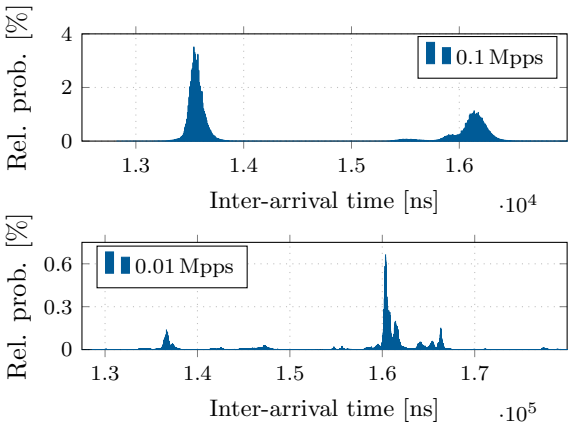


Figure 7: trafgen precision

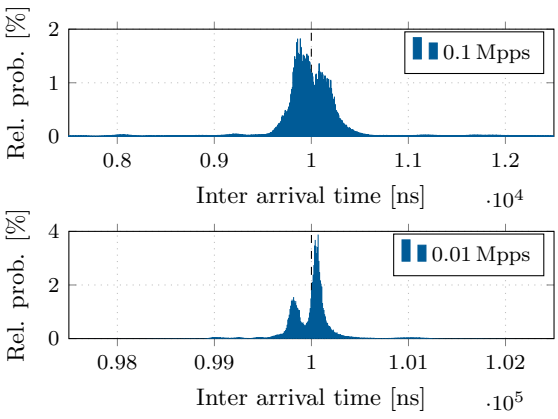
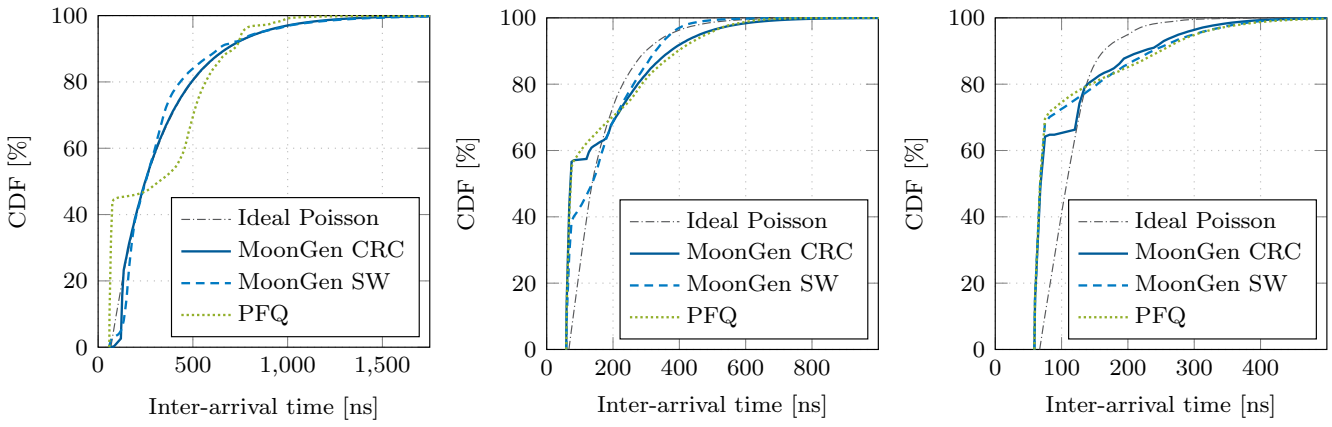


Figure 8: D-ITG precision



(a) 3 Mpps (1 thread)

(b) 6 Mpps (2 threads)

(c) 9 Mpps (3 threads)

Figure 9: Precision of Poisson traffic generation

packets are sent in bursts at 0.1 Mpps, compared to less than 0.01% at 0.01 Mpps. We categorize D-ITG as accurate and precise at very low rates but not performant.

5.5 Precision with Poisson traffic pattern

This section investigates the traffic generation precision when a non-CBR traffic pattern is required. In particular, we study the generation behavior when the inter-packet gap is set to be a Poisson process. The tests presented in this section allow studying the impact of the cooperation of multiple threads on the generation precision. The previous tests have been restricted to a single thread transmitting packets to reliably generate CBR traffic. This is no longer the case when using a Poisson process: overlaying several independent Poisson processes forms a new Poisson process. Using multiple threads to increase the performance is trivial and allows overcoming performance restrictions. Although real traffic resembles a self-similar pattern [24], traffic generated by a Poisson process can approximate the self-similar pattern for short time spans, e.g., in a synthetic benchmark [26]. Moreover, self-similar patterns are not implemented by any readily available software packet generator. Poisson is implemented in D-ITG, pfq-gen, and MoonGen. We skip D-ITG here as we are interested in high packet rates.

We measure a maximum throughput of 12.1 Mpps with pfq-gen on 4 threads and 12.9 Mpps with MoonGen software rate control on 3 threads. This corresponds to the previously measured limit for unbatched transmits; adding another core does not improve the throughput. The CRC-based rate control is able to generate any configured rate as it can use batches consisting of valid and invalid packets internally.

Figure 9 shows how the achieved precision degrades as the rate and number of threads being used increases from 3 Mpps to 9 Mpps and 1 to 3 respectively. This measurement highlights a shortcoming of the CRC-based implementation: it cannot represent all possible gaps due to minimum packet sizes of the invalid packets. Some NICs like the Intel XL710 NICs pad short packets to the minimum size of 64 bytes while others (82599) allow smaller sizes. However, illegally short packets can be troublesome for the device under test: we experienced irregular behavior when the NetFPGA interface controller was receiving such packets.

The higher the requested throughput and the number of threads being used, the lower the precision of the generation: overlaying Poisson processes is imperfect in practice. This technique assumes that the Poisson processes cannot influence one another. However, this is not the case. A packet incurs a queuing delay if a thread tries to send a packet while a packet by another thread is still being transmitted. This effect is visible at both 6 Mpps and 9 Mpps, it leads to more bursts and fewer packets with larger inter-arrival times than analytically expected.

5.6 Lessons learned

We have compared three methods for packet generation:

The **hardware supported approach** offers high performance and reasonable accuracy. However, it shows low precision on the investigated NICs as they generate small bursts instead of CBR traffic. Further, it is inflexible as it cannot be used for arbitrary distributions.

The **pure software approach** can offer high accuracy as long as overloading does not occur. Either high performance or high precision are achievable, depending on the allowed burst size leading to the previously presented issues (cf. Section 2). Differences in performance and precision between the IO frameworks are visible at high packet rates, with the DPDK-based frameworks usually performing better.

The **corrupted CRC approach** offers high performance, high accuracy, and high precision. Despite its advantages, this method requires setups that can withstand the large number of invalid packets used as fillers.

Trafgen and D-ITG are still not obsolete, despite the advantages of modern software packet generators. D-ITG is ideal for environments that do not require high packet rates. It is precise and features a large set of traffic types and patterns. The main advantage is that it works on any NIC supported by Linux – the other investigated packet generators rely on specialized drivers only available for certain NICs.

CPU	Pkt. Gen.	Throughput
Ivy Bridge @ 3.5 GHz	MoonGen	12.9 Mpps
	pfq-gen	7.2 Mpps
Ivy Bridge @ 1.6 GHz	MoonGen	6.8 Mpps
	pfq-gen	3.6 Mpps

Table 4: Throughput at different clock frequencies

CPU [GHz]	MSE [ns^2]
1.6	29318
1.7	28364
1.9	24384
2.2	20185
2.5	16671
(Turbo) 3.5	15237

Table 5: MoonGen: MSE at different CPU clock frequencies (2 Mpps)

6. INFLUENCE OF DIFFERENT CPU MICROARCHITECTURES

Packet generation is a highly demanding task for the CPU. This section investigates the impact of both CPU microarchitecture and clock speed on this process.

6.1 Throughput vs. CPU frequency

Table 4 shows the impact of clock frequency over the throughput for pfq-gen and MoonGen (pure software approach for rate control). We selected the latter as an example of the DPDK userspace driver against one with a patched kernel driver as found in pfq, netmap and PF_RING ZC. The experiment is executed on an Intel Xeon E3-1265L v2 with a maximum frequency of 3.5 GHz with turbo-boost and repeated with the CPU manually throttled to 1.6 GHz.

Both packet generators react similarly: the throughput scales well with the CPU frequency. The sub linear behavior of the maximum throughput at 12.9 Mpps is due to hardware limits for unbatched transmission identified in Section 5.4.3. These results are expected as high-performance IO frameworks are known to scale linearly with the CPU frequency [19, 25]. Therefore, we consider the linear scaling as a property of the underlying frameworks rather than a property of the packet generators themselves.

6.2 Precision vs. CPU frequency

The following tests concentrate only on the MoonGen traffic generator, as it proved to be the most precise at rates of 2 Mpps and above. We configure the software packet rate control to generate 2 Mpps to quantify how the CPU clock frequency influences the precision. Previous measurements show this rate is well below the generation limit for any clock frequency, i.e., enough processing cycles are available to cope with the task. However, despite the availability of enough clock cycles to fulfill the task, a difference in the MSE of the packet distribution is visible in Table 5. The error decreases as the CPU frequency increases, i.e., faster CPUs achieve a higher precision, even if the CPU power is not needed for the desired rate.

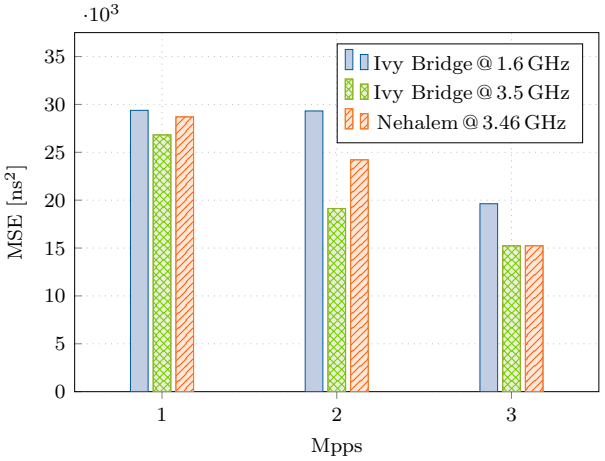


Figure 10: MoonGen: comparison between microarchitecures, throughput and CPU frequency

6.3 Precision vs. CPU microarchitecture

The third investigation involves two different CPU microarchitectures: the Ivy Bridge released in 2012 (Intel Xeon E3-1265L v2) and the Nehalem released in 2008 (Intel Core i7-960). We generate 1, 2, and 3 Mpps with MoonGen's software rate control and measure the precision as MSE in Figure 10. As with the previous results in Section 5.4.3, the MSE improves for the software rate control with increasing rates. This happens because the distance between the target and the minimum possible gap decreases and accidental bursts are less severe for this metric. There is only a small difference between the analyzed microarchitectures when clock frequencies are almost identical. The clock frequency has a larger impact on the precision than the microarchitecture itself.

6.4 Lessons learned

The measurements in this section show the importance of the clock frequency for performance and precision. Additionally, there is only a minor influence of the microarchitecture on the precision, at least the investigated ones. The higher the CPU frequency, the better performance and precision. With lowering silicon costs and rising consumer needs, manufacturers push one of two things: clock speed or core count. In particular, higher CPU frequencies entail lower number of cores for the CPU, making a higher clock speed more attractive when a CBR traffic needs to be generated. A higher core count is attractive for more complex scenarios that can be parallelized, i.e., poisson traffic generation.

7. PACKET GENERATORS FOR LATENCY MEASUREMENTS

Packet generators are not only used to precisely and accurately generate traffic, but they might be also helpful to derive useful metrics about the device under test: throughput and latency. Throughput can be measured by counting the number of packets being successfully processed by the device under test, latency requires more elaborated methods. In particular, measuring the latency requires timestamping the exact point in time a packet is sent or received.

7.1 Approaches for measuring latency

We identify three different types of timestamping methods.

Software timestamping without framework support is the easiest implementation: the software simply takes a timestamp before sending and after receiving a packet to/from a high-level interface like socket APIs. Potential problems are context switches and queuing delays as the packet crosses the boundary between the program and the driver in the kernel. Linux allows offloading the timestamp to the kernel via a socket option to alleviate these problems.

Software timestamping with framework support takes the timestamp at the framework or driver level. The low-level nature of packet IO frameworks helps. For example, DPDK moves the whole driver into the same process as the packet generator. A packet generator based on it can thus precisely control when a packet is sent and take the transmission timestamp in a precise manner. Reception typically works in a busy-waiting loop, polling the NIC for new packets as often as every 100 CPU cycles.

A third and most reliable solution is **hardware timestamping**. This moves the timestamping process as close as possible to the physical layer thus eliminating further sources of error (i.e., CPU scheduling, driver overhead, or PCIe transfer).

7.2 Evaluated metrics

Latency measurement features are rare in packet generators. To the best of our knowledge, and considering the packet generators being analyzed in this paper, only D-ITG and MoonGen provide support for latency analysis. MoonGen uses hardware timestamping as it relies on hardware capabilities present on Intel NICs. D-ITG implements timestamping without framework support, as it uses traditional APIs without making use of the timestamp offloading available in Linux. We also implemented a version of MoonGen with pure software timestamping to evaluate the performance of software timestamping with a fast IO framework.

Latency measurement is affected by both a systematic error and a random error. The former is caused by deterministic delays through processing, the latter from the time spent in buffers and resource contention from concurrent tasks. To minimize both of them, the timestamp should be moved as close as possible to the actual physical transmission or reception of a packet. Without hardware support, the timestamps can only be taken in software and queuing delay can cause high jitter. Deterministic processing steps contribute to the systematic error, providing a fixed offset on the final timestamp.

The proposed tests aim to evaluate the **accuracy** (i.e., average of the measured values) and the **precision** (i.e., standard deviation) of latency measurements when one of the three approaches is being used. An inaccurate timestamping mechanism has little impact on the results: the systematic error can be determined and subtracted from latency measurements. Poor precision is more problematic as it limits the usefulness of the resulting data. While it is still possible to determine the average latency if the precision is poor, it is difficult to estimate important characteristics of the device under test (e.g., buffer size). In addition, statistical parameters such as maximum latency or 99th percentile cannot be evaluated properly when the precision is poor.

The third metric we take into account is the **granularity** of the packet generator itself. Software using RDTSC has a granularity of the CPU's clock frequency, i.e., typically less than 1 ns. `clock_gettime()` has a granularity of 1 ns on modern systems with proper arguments as it internally relies on RDTSC as well. Hardware-assisted solutions as implemented in MoonGen depend on the NIC being used. The Intel 82599 offers a granularity of 12.8 ns [20], older 1 Gbit/s NICs often support only 64 ns [21].

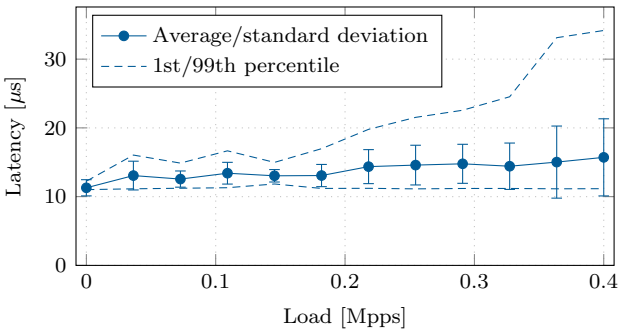


Figure 11: Precision of software timestamping without framework support

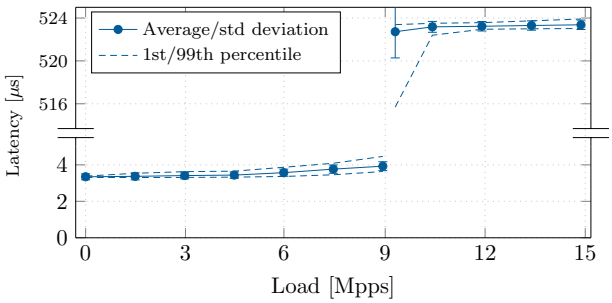


Figure 12: Precision of software timestamping with framework support (MoonGen)

7.3 Evaluation

In this section we evaluate the three different approaches using an external loopback connection of a NIC port with itself via a short (≈ 10 cm/2.5 in) fiber cable as test setup. As D-ITG would require a second host echoing the traffic back, we emulate its behavior by implementing a tool that performs software timestamping without framework support on raw sockets.

7.3.1 Hardware

We rely on MoonGen working with hardware support (Intel 82599 NIC) to evaluate the latency in this scenario. This implementation offers a granularity of 12.8 ns on the Intel 82599 NIC and reports latencies between 294.4 ns and 320 ns (23 to 25 units of measurement) depending on the packet rate. The latency never varies by more than 12.8 ns for a given packet rate. We use this result as our ground truth, i.e., the maximum achievable precision and accuracy.

7.3.2 Software without framework support

Figure 11 shows how the precision of our implementation of software timestamping without fast IO framework

support deteriorates as we apply an increasing load of background traffic using trafgen[2]. These rates are approximate due to the poor accuracy of trafgen. However, the graph shows a trend: the standard deviation is between 1 μs and 2.4 μs below 0.2 Mpps and then increases up to 5.6 μs under full load.

7.3.3 Software with framework support

Software timestamping with fast IO framework support in MoonGen performs better by an order of magnitude as visualized in Figure 12. The standard deviation increases from 0.16 μs to 0.24 μs between no background load and 9 Mpps. However, a higher load causes a sudden increase in both average latency and standard deviation. Rates above 9.3 Mpps show standard deviations of $\approx 0.5\,\mu$s. High background traffic causes this software timestamping method to become inaccurate while still staying reasonably precise. It is, however, both imprecise and inaccurate in the area between 9 Mpps and 9.3 Mpps. The reason for this increase remains unclear. We believe it is caused by the hardware because only the accuracy suffers by factor of about 200 while the precision is only affected by a factor of 2.

We conducted this experiment with hardware rate control on an Intel 82599 NIC for the background traffic to achieve the highest possible rate. Using the best rate control method identified earlier (i.e., CRC) requires loading the NIC with full line at all times and hence suffers from poorer precision and accuracy even at lower rates.

7.4 Lessons learned

The test campaign performed in this section assess the precision of software timestamp (with and without IO framework). We use hardware solution as our ground truth because it moves the timestamping process as close as possible to the physical layer thus eliminating further sources of error.

We distinguish two measurement scenarios with different requirements for the benchmarking precision: software devices and hardware devices. The former has lower requirements as the experiments deal with higher latencies (in the range between 10 μs and 100 μs). For example, our measurements in Section 2 exhibit latencies between 14 μs and 110 μs. Based on this, we derive a precision requirement of 1 μs to benchmark software devices. This means both software timestamping with framework support and hardware timestamping are precise enough for this task. Hardware devices, on the other hand, exhibit latencies in the order of hundreds of nanoseconds and consequently need a packet generator with a precision lower than 100 ns. Only hardware timestamping can be used in this case.

The feasibility of benchmarking software devices also depends on the measured metric. It is always possible to determine the average latency if the characteristics of the packet generator are known. Histograms of observed latencies, that can provide further insight into internals of a system, can only be measured with framework support or hardware timestamping. Measuring the maximum latency of a system is limited by the worst-case behavior of the packet generators: it is impossible to know whether an outlier in an experiment comes from the packet generator or the tested system. Timestamping with framework support shows outliers of up to 5 times the average value, while timestamping without framework support has outliers up to 60 times the average value.

Experimenters should calibrate their packet generators before conducting experiments involving any latency measurements. The reliability of the packet generator can be checked by running the test on a setup where the tested system is replaced with a simple cable. Not only precision is important: the accuracy or systematic error is in the microsecond-range with the tested approaches and can thus contribute a significant part of the overall latency. Measuring it beforehand allows eliminating this error, i.e., the accuracy of a packet generator does not matter in a well-designed experiment with a calibrated packet generator.

[2]Chosen because it achieves the highest rate without requiring exclusive access to the NIC.

8. RELATED WORK

The performance and precision of software packet generators has been subject of previous studies. In particular, Paredes-Farrera et al. [23] in 2006 analyzed the accuracy of packet generators. They found that timing primitives available in Linux limit the achievable precision. Polling techniques can be precise but at the same time are massively affected by the current CPU load of the system. In 2010 Botta et al. [13] performed a comparison between software packet generators. In particular, they investigate the inter-packet gap precision when the traffic being generated follows different distributions. They showed that despite meeting the required bandwidths, the actual distribution of the generated traffic can differ substantially from the expected pattern impacting the measured results. Both of the aforementioned papers investigate only traffic rates below 1 Gbit/s.

With the availability of high-performance IO frameworks, traffic rates of 10 Gbit/s are easily possible [18, 25]. Bonelli et al. [11] described a system for precise traffic generation in 2012, pfq-gen evaluated by us is a successor of this system. It is designed to support packet generation with CBR or according to a Poisson process. The performed measurements show high throughput but the precision has only been validated with a software solution, limiting the precision of the measurement. The packet generator MoonGen [18] relies in its evaluation process on the hardware timestamping feature available on commodity Intel 1 Gbit/s. Nowadays, accurately evaluating the precision at higher rates with commodity hardware is not possible, as Intel's 10 Gbit/s commodity NICs can only sample timestamps of a subset of the received packets.

To this end, we performed our measurement campaign using OSNT [10]: an open source hardware based traffic generator and monitoring system based on the NetFPGA [29]. The system offers the ability to timestamp packets at line rate for traffic of 10 Gbit/s, thus enabling an accurate estimation of the trade-off between throughput and precision in today's solutions.

9. CONCLUSION

This paper proposes a comparison between different approaches in packet generation. Although previous papers already have proposed a similar comparison, we investigate the software packet generators in the light of new developments such as 10 Gbit/s Ethernet, high-performance IO frameworks and modern CPUs. While the new generation

of high performance software packet generators achieves the required performance, precision problems appear for packet rates above 1 Mpps. Hardware rate control features found on Intel commodity NICs proved to be imprecise as well.

MoonGen using the rate control implemented with the corrupted CRC approach proved to be the most precise solution. However, it depends on the capabilities of the device under test to handle the corrupted frames. Increasing the performance can also be done by scaling the generation process with multiple cores. In this case, the CBR traffic cannot be anymore generated reliably. On the other hand, Poisson generation might take advantage of its main feature: overlaying several independent Poisson processes forms a new Poisson process. Given the trade-off between number of cores and per-core frequency, it is possible to get the most out of the machine with Poisson traffic.

The fixation on CBR traffic arguably stems from the definition of constant load as CBR in RFC 1242 [15]. Even this standard from 1991 notes that it is an unrealistic traffic pattern. Hence, we argue to consider a Poisson distribution instead of CBR traffic: Poisson traffic is both easier to generate in a performant and reliable manner and a more realistic test case.

Finally, fast IO frameworks can also be used for timestamping when benchmarking software forwarding devices. Hardware timestamping solutions should be preferred in general if available.

Acknowledgments

The authors would like to thank Nicola Bonelli for insightful discussions about PFQ and Dominik Schöffmann for helping us with PFQ test setups. This paper is jointly supported by the European Union's Horizon 2020 research and innovation program 2014-2018 under the SSICLOPS (grant agreement No. 644866) and ENDEAVOUR (grant agreement No. 644960) projects and the BMBF projects DecADe (16KIS0538) and SENDATE-Planets (16KIS0472). This paper reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

10. REFERENCES

[1] D-ITG. http://traffic.comics.unina.it/software/ITG/index.php. Last visited: 2017-01-04.

[2] DPDK. http://www.dpdk.org/. Last visited: 2017-01-04.

[3] NetFPGA Official Website. http://www.netfpga.org. Last visited: 2017-01-04.

[4] netmap. https://github.com/luigirizzo/netmap. Last visited: 2017-01-04.

[5] ntop official website. http://www.ntop.org/. Last visited: 2017-01-04.

[6] PFQ. https://github.com/pfq/PFQ. Last visited: 2017-01-04.

[7] PF_RING ZC. https://github.com/ntop/PF_RING.git. Last visited: 2017-01-04.

[8] Pktgen-DPDK. http://dpdk.org/browse/apps/pktgen-dpdk/refs/. Last visited: 2017-01-04.

[9] trafgen. http://netsniff-ng.org/. Last visited: 2017-01-04.

[10] G. Antichi, M. Shahbaz, Y. Geng, N. Zilberman, A. Covington, M. Bruyere, N. McKeown, N. Feamster, B. Felderman, M. Blott, et al. OSNT: Open Source Network Tester. *Network*, 28(5), 2014.

[11] N. Bonelli, A. Di Pietro, S. Giordano, and G. Procissi. Flexible High Performance Traffic Generation on Commodity Multi–Core Platforms. In *International Conference on Traffic Monitoring and Analysis (TMA)*. Springer, 2012.

[12] 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 conference (PAM)*. Springer, 2012.

[13] A. Botta, A. Dainotti, and A. Pescapé. Do You Trust Your Software-Based Traffic Generator? *Communications Magazine*, 48(9), 2010.

[14] A. Botta, A. Dainotti, and A. Pescapè. A tool for the generation of realistic network workload for emerging networking scenarios. *Computer Networks*, 56(15), 2012.

[15] S. Bradner. Benchmarking Terminology for Network Interconnection Devices. RFC 1242 (Informational), July 1991.

[16] S. Bradner and J. McQuaid. Benchmarking Methodology for Network Interconnect Devices. RFC 2544 (Informational), March 1999.

[17] P. Emmerich and contributors. MoonGen. https://github.com/emmericp/MoonGen. Last visited: 2017-01-04.

[18] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *Internet Measurement Conference (IMC)*. ACM, 2015.

[19] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle. Comparison of frameworks for high-performance packet IO. In *Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE/ACM, 2015.

[20] Intel Corporation. *Intel 82599 10 Gigabit Ethernet Controller Datasheet Rev. 2.9*, January 2014.

[21] Intel Corporation. *Intel 82580EB/82580DB Gigabit Ethernet Controller Datasheet Rev. 2.7*, September 2015.

[22] Intel Corporation. *Intel Ethernet Controller X550 Datasheet Rev. 2.5*, March 2016.

[23] M. Paredes-Farrera, M. Fleury, and M. Ghanbari. Precision and accuracy of network traffic generators for packet-by-packet traffic analysis. In *Testbeds and Research Infrastructures for the Development of Networks and Communities (TRIDENTCOM)*. IEEE, 2006.

[24] V. Paxson and S. Floyd. Wide-Area Traffic: The Failure of Poisson Modeling. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 1994.

[25] L. Rizzo. netmap: a novel framework for fast packet I/O. In *Annual Technical Conference (ATC)*. USENIX, 2012.

[26] T. G. Robertazzi. *Computer networks and systems: queueing theory and performance evaluation, Chapter 7.6: Self-Similar Traffic*. Springer Science & Business Media, 2012.

[27] P. Saul. Direct digital synthesis. In *Circuits and Systems Tutorials*, 1996.

[28] M. Tahhan, B. O'Mahony, and A. Morton. Benchmarking Virtual Switches in OPNFV. draft-ietf-bmwg-vswitch-opnfv (Internet-Draft), October 2016.

[29] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. NetFPGA SUME: Toward 100 gbps as research commodity. *Micro*, (5), 2014.