Advanced Computer Networking (ACN)

IN2097 – WiSe 2024–2025

Prof. Dr.-Ing. Georg Carle, Sebastian Gallenmüller

Christian Dietze, Paul Emmerich, Max Helm,
Benedikt Jaeger, Marcel Kempf, Jihye Kim, Patrick Sattler

Chair of Network Architectures and Services
School of Computation, Information, and Technology
Technical University of Munich

Performance Measurements

Performance Measurements

Performance metrics

Performance metrics for network devices

Metrics (taken from RFC 2544):

- **Throughput**: bandwidth and packet rate
- **Latency**: average, standard deviation, median, jitter, percentiles
- Other metrics:
  - Frame loss rate
  - Maximum burst length
  - Recovery after overload
  - Recovery after system reset

These metrics be measured under varying circumstances: type of traffic, applied load, device settings, ...

Performance metrics

Common standards for performance evaluations

RFC 2544 - Benchmarking Methodology for Network Interconnect Devices

Used by almost all vendors of network devices. Published in 1999, does not take effects of modern software-based systems into account.

RFC 1242 - Benchmarking Terminology for Network Interconnection Devices

Defines basic terminology for benchmarks, e.g., constant load is defined as packets send in a fixed interval.

RFC 8204 - Benchmarking Virtual Switches in OPNFV

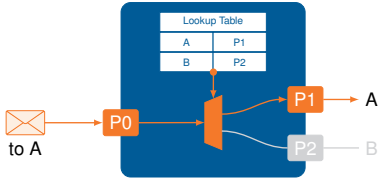
Describes a modern framework to measure virtualized network devices.

IMIX - Internet MIX

Distribution of packet sizes found on the Internet. Often used by firewall vendors to quantify throughput. Not an official standard, published by Agilent.

Performance Measurements

Throughput



Network devices performing only basic functions like switching and routing are usually only limited by:

- Line rate
- Size/speed of the lookup-tables:
  - Lookup in hardware: content-addressable memory (CAM)
    → small, fast, and expensive
  - Lookup in software: RAM, CPU cache
    → big, cheap, and slow

Interesting scenarios:

- Complex processing, e.g., in firewalls or SDN switches
- Software routers
- Virtual switches to connect virtual machines (VMs) to physical devices

Throughput

Bandwidth vs. packet rate

Simply measuring the throughput in Gbit/s is not sufficient:

- Simple packet processing tasks (switching, routing) only process packet headers
- Actual size of the packet only plays a minor role
- **Exception:** Applications processing payload of packets (e.g., payload encryption), there the size of the packets matters

Packet processing costs:

Processing a packet has an inherent cost independent of its size

Worst-case scenario for network devices:

- Network traffic at line rate
- Minimum-sized packets

Hardware vendors often claim to reach line rate without mentioning the used packet size.

Throughput

What's the maximum packet rate of my network?

How many packets per second can you send over a 10 Gbit/s Ethernet connection?

- Common misconception: minimum size of an Ethernet frame is 64 bytes, so about 20 Mpps

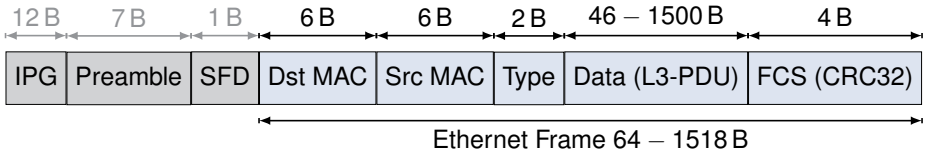
Throughput

What's the maximum packet rate of my network?

How many packets per second can you send over a 10 Gbit/s Ethernet connection?

- Common misconception: minimum size of an Ethernet frame is 64 bytes, so about 20 Mpps

- That does not take the physical layer into account
- Ethernet frames (Layer II) are encapsulated in Ethernet packets (Layer I)
- Ethernet packets start with a 7 byte preamble and 1 byte start-of-frame delimiter
- Minimum interpacket gap (IPG, also incorrectly referred to as interframe gap) of 12 bytes
- Minimum packet size: 84 bytes
- Maximum packet rate: 14.88 Mpps



Throughput

Measuring Throughput

Simplest methodology

- Configure device under test (DuT) to forward packets from ports A to ports B
- Apply the highest possible packet rate on ports A of DuT
- Measure the packet rate on ports B of DuT

Problems

- Potentially overloads the DuT, leading to different behavior
- The achieved throughput is not necessarily the highest possible throughput
- When also measuring latency: full buffers lead to worst-case latency

Typical two-node measurement setup

Throughput

Methodology from RFC 2544

- Configure DuT to forward packets from ports A to ports B
- Apply varying rates on ports A
- Find the highest rate at which no packet loss occurs, e.g., by doing a binary search

Typical two-node measurement setup

Problems

- RFC 2544 calls for a completely loss-free test run
- Some devices lose packets when suddenly faced with a high packet rate due to power-saving features
- Benchmarks take a long time, especially when testing a lot of configurations or traffic types

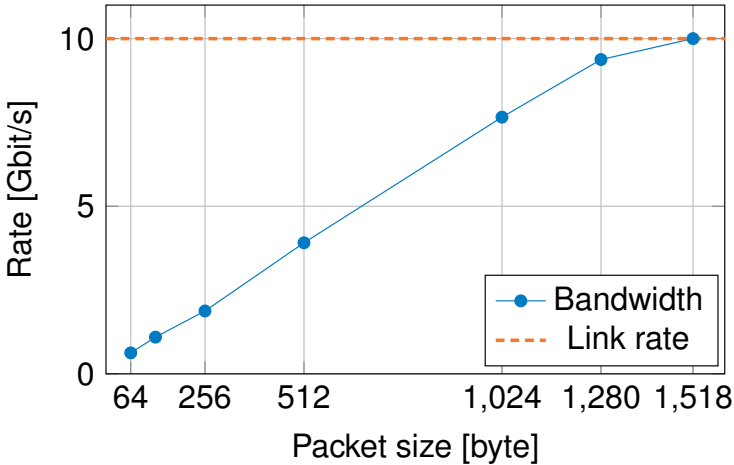
The best methodology depends on the DuT

- Test the behavior of the DuT and adapt the methodology: applying the maximum rate to measure the throughput is often sufficient
- Allow a small packet loss at the beginning or slowly ramp up the rate
- Reduce test time (RFC 2544 recommends at least 60 seconds)

Throughput

Example: Linux Router, Bandwidth

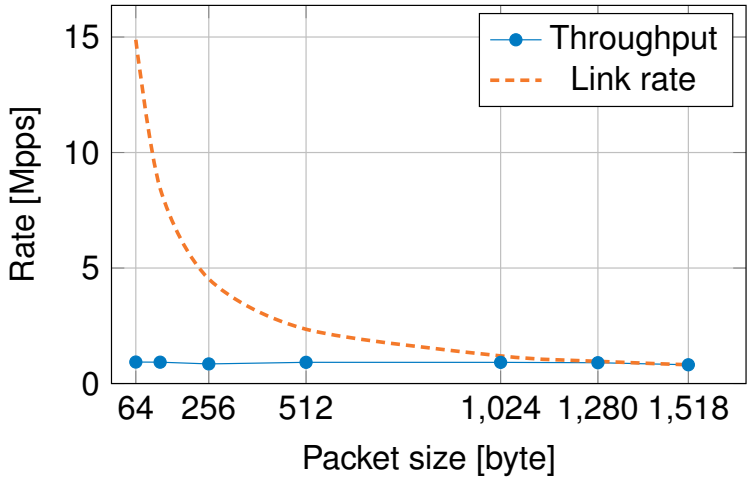
- Maximum throughput determined by an RFC 2544 test
- Linear increase with packet size, i.e., the size does not matter



Throughput

Example: Linux Router, Packet Rate

- Performance leaves room for improvement with small packets



Example: Comparison

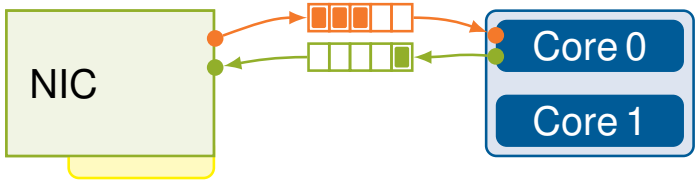
Forwarder	Throughput [Mpps]
Line rate at 10 Gbit/s	14.88
Open vSwitch (Linux)	1.88
Linux routing	1.58
FreeBSD routing	1.30
Linux bridging	1.11

- This and the previous tests were using only a single 3.3 GHz CPU core
- Forwarding is a parallelizable problem
- No dependencies between packets of different flows
- How to parallelize?

Performance Measurements

Parallel Packet Processing

Architecture of Network Cards



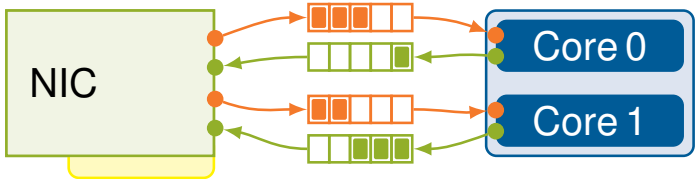
The Network Interface Card (NIC) offers:

- Interface for incoming transfer (RX queue)
- Interface for outgoing transfer (TX queue)

Problems:

- Only one core can access NIC simultaneously
- Scalability of multithreaded network applications very poor
- Single core performance limits throughput of whole system

Architecture of Modern Network Cards



Modern NICs have multiqueue support:

- High number of queues (e.g., 1536 queue pairs on Intel XL710 10/40 GbE NICs)
- Can be used completely independent from each other
- A queue is typically used by a CPU core exclusively to improve cache locality
- Allows for perfect linear multi-core scaling

Parallel Packet Processing

Incoming traffic is distributed

- On a per-packet basis
  - Every packet may be processed by a different core
  - \- Slow if protocol state synchronization necessary between cores
  - \- May cause packet reordering
- On a per-flow basis via hashing over protocol, addresses and ports
  - One flow is only processed by a specific core
  - \+ Fast if protocol state resides in the cache of the according core
  - \+ Prevents packet reordering within a flow
- Explicitly configured filters
  - Flows can be mapped to cores explicitly
  - \- Hashing leads to better balancing
  - \+ Useful to forward traffic to virtual machines

Parallel Packet Processing

Multiqueue support influences application design:

- All threads should be pinned
- Threads only handle specific flows
- This concept does not map the POSIX socket API (1983)

Userspace vs kernelspace

Packet forwarding is typically done in the kernel to improve performance.

Packet processing in the kernel

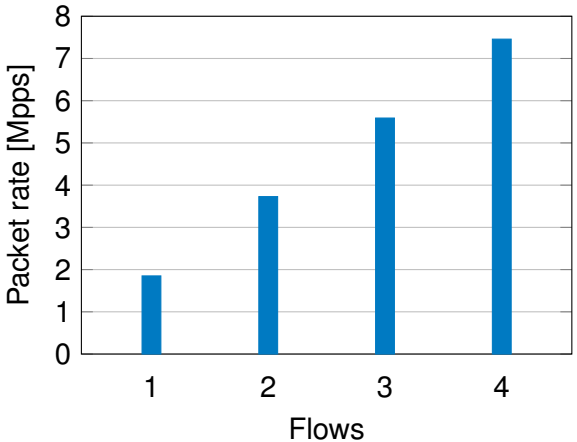
- Low-level driver interface in Linux: NAPI (2001)
- NIC uses different interrupts (IRQs) for different queues
- IRQs can be pinned to CPU cores, thus achieving multi-core scaling

- All previous examples were kernel-based forwarders
- In-kernel forwarding is one order of magnitude faster than the socket API
- We do not discuss socket-based performance here

Parallel Packet Processing

Example: Multi-core Scaling via Hashing

- Example: forward UDP packets with Open vSwitch
- Generate different flows by using multiple UDP source ports
- Linear scaling up to the number of (physical) CPU cores



Performance Measurements

Improving Throughput

Bottlenecks

CPU processing power

Yes: bottleneck in the previous tests

NIC processing power

Sometimes:

- Cheap consumer NICs can often manage less than 500 kpps[1]
- Intel XL710 (2x 40 Gbit/s) achieves only 42 Mpps (28 Gbit/s at 64 byte frames)

Bus (PCIe) bandwidth

PCIe 3.0 is 8 Gbit/s per lane. E.g., the Intel XL710 dual port 40 Gbit/s NIC only has 8 lanes.

Memory bandwidth

No: modern PCIe devices can access the CPU's cache directly

CPU caches

Sometimes: depends on the forwarding software and use case

[1] cf. https://github.com/luigirizzo/netmap/blob/b249e9cd5ae12c7d57f3d4b5dc4aa8edded924ad/LINUX/README

Improving Throughput

Profiling

- Fastest in-kernel forwarder: 1.88 Mpps at 3.3 GHz (single core)

$$\frac{3.3\,\text{GHz}}{1.88\,\text{Mpps}} = 1755\,\frac{\text{Cycles}}{\text{Pkt}}$$

- What is taking 1755 cycles to process a packet?
- Using profiling (`perf` on Linux) to find out
- `perf` allows us to see CPU cycles spent in each function

Improving Throughput

Profiling

- Fastest in-kernel forwarder: 1.88 Mpps at 3.3 GHz (single core)

$$\frac{3.3\,\text{GHz}}{1.88\,\text{Mpps}} = 1755\,\frac{\text{Cycles}}{\text{Pkt}}$$

- What is taking 1755 cycles to process a packet?
- Using profiling (`perf` on Linux) to find out
- `perf` allows us to see CPU cycles spent in each function
- Thousands of functions involved, so categorize them

Improving Throughput

Receive
Receiving packets from the NIC, including interrupt overhead

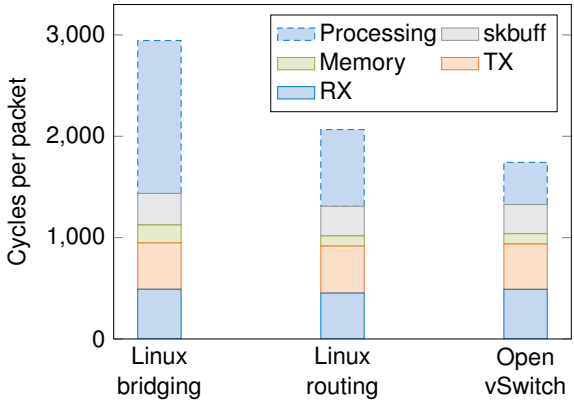
Transmit
Sending packets to the NIC

skbuff
`skbuff`s are kernel data structures containing packet data and metadata, handling them comes with overhead

Memory
Memory management overhead

Processing
Reaching a forwarding decision

Profiling results



- About 1000 cycles just to receive and send a packet
- 400 cycles to manage memory and `skbuff` overhead

Improving Throughput

Can we do better?

Shift packet processing entirely into the user space:

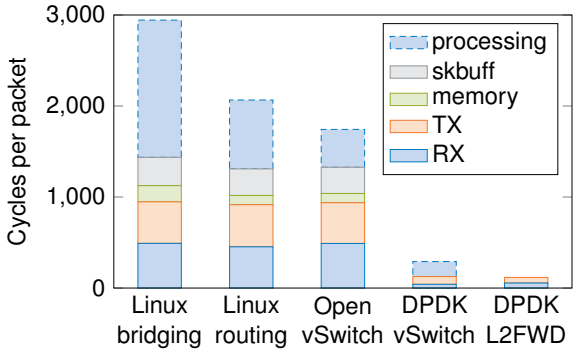
- Map DMA buffers into a user space application
- Bypass the entire network stack of the OS

+ Fewer expensive system calls
+ Simplified memory management and data structures
+ Faster than the kernel by an order of magnitude
+ Batch processing throughout the whole application
- Handles only raw packets, protocols need to be implemented in the application
- NIC used exclusively by a single application
- Not API-compatible with traditional user space applications

- Examples:
  - DPDK
  - netmap
  - Snabb

Improving Throughput

DPDK vs. the Linux kernel



- Open vSwitch can use DPDK for packet IO instead of running in the kernel
  - 291 cycles per packet
- DPDK L2FWD forwards packets without consulting a lookup table
  - 117 cycles per packet

Performance Measurements

Sources of Latency

Source of delay		
Serialization (10 Gbit/s)	10	bit / ns
Propagation ($\approx \frac{2}{3}c$)	0.2	m / ns
Calculation (3 GHz CPU)	3	cycles / ns

- Serialization is given by the Ethernet standard
- Propagation is determined by the length of cable
- Reasonable calculations with around 3000 cycles only takes 1 μs
- Time spent for **buffering** can be in the range of several μs

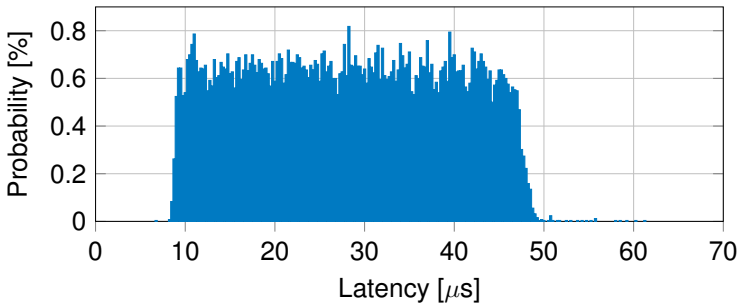
Buffers matter, processing time often does not.

Latency

Techniques for packet reception

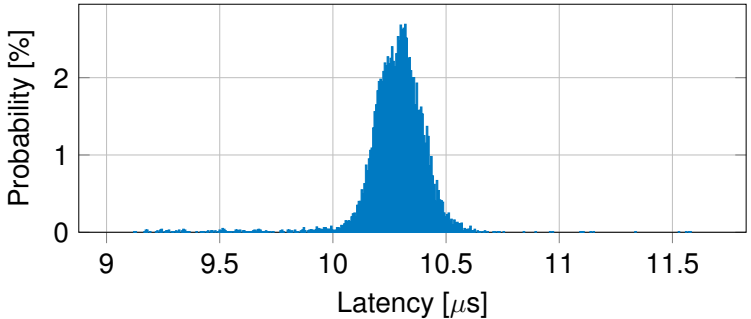
- Classic system one interrupt per packet
  - \+ low latency
  - \- low throughput (interrupts are expensive)
- More modern systems process several packets per interrupt
  - \+ high throughput (lower costs per packet)
  - \- high latency
- No interrupts at all
  - \+ low latency depending on polling frequency
  - \+ high throughput
  - \- inefficient at low packet rates (busy waiting for packets)

Latency with Batching



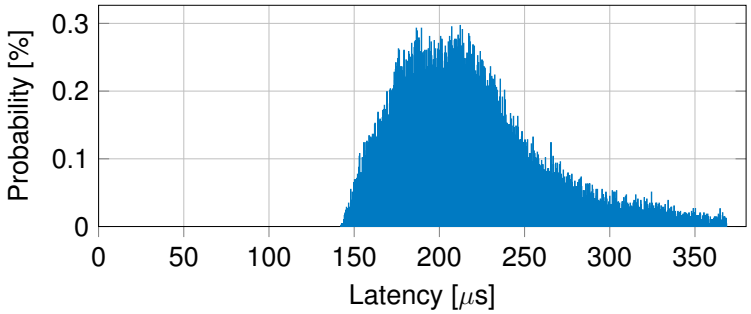
- Open vSwitch on Linux
- Batch processing
- Roughly a uniform distribution

Latency with Polling/Busy Wait



- DPDK on Linux
- Very small batches
- Roughly a normal distribution

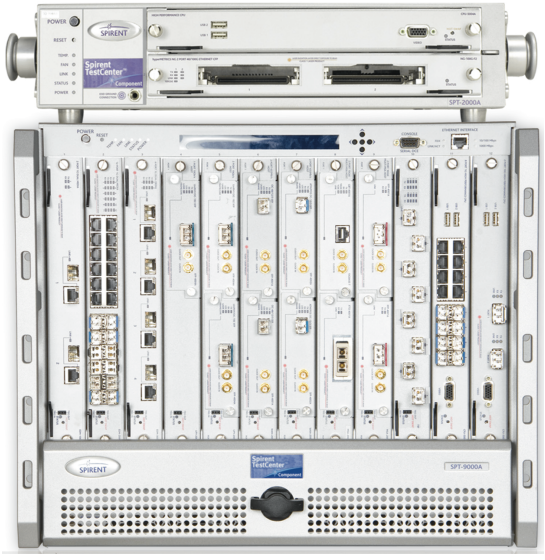
Latency through a VM



- Forwarding through a VM
- Several buffers
- A long tail distribution, i.e., some packets are significantly slower than the average

Performance Measurements

Packet Generators



Source: www.spirent.com

Packet Generators

Properties of packet generators

- Hardware packet generators are
  - Precise
  - Accurate
  - Fast

Packet Generators

Properties of packet generators

- Hardware packet generators are
  - Precise
  - Accurate
  - Fast
- Software packet generators
  - Run on cheap commodity hardware
  - Flexible

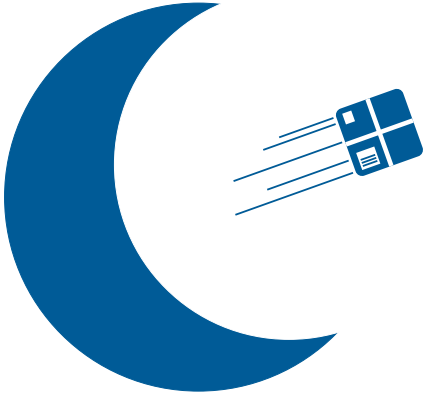
Packet Generators

Properties of packet generators

- Hardware packet generators are
  - Precise
  - Accurate
  - Fast
- Software packet generators
  - Run on cheap commodity hardware
  - Flexible
- Key challenges for software packet generators
  - Rate control
  - Timestamping

Packet Generators

MoonGen



- MoonGen is a software packet generator developed here
- Available as open source software at `https://github.com/emmericp/MoonGen`

Packet Generators
MoonGen

Design goal of MoonGen

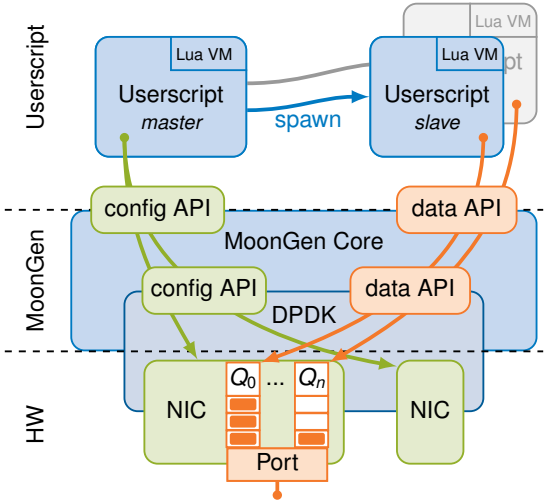
Combine the advantages of hardware and software packet generators while avoiding their disadvantages.

Design goal of MoonGen

Combine the advantages of hardware and software packet generators while avoiding their disadvantages.

- Fast: DPDK for packet I/O, explicit multi-core support
- Flexible: Craft all packets in user-controlled Lua scripts
- Timestamping: Utilize hardware features found on modern commodity NICs
- Rate control: Hardware features and a novel software approach

Architecture

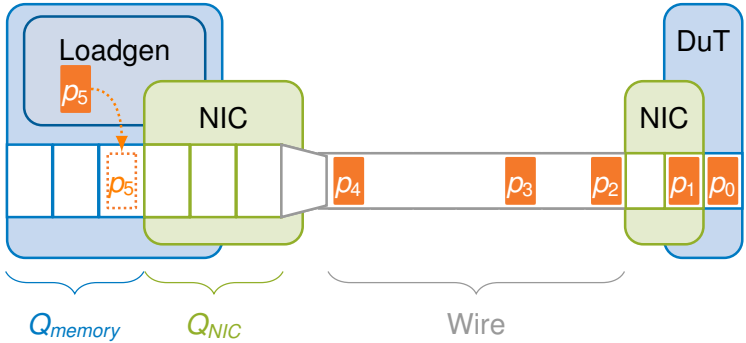


Hardware timestamping

- NICs support Precision Time Protocol (PTP) for precise clock synchronization
- PTP support requires hardware timestamping capabilities
- These can be (mis-)used for delay measurements
- Typical precision
  - $\pm$6.4 ns (Intel 10 GbE chips)
  - $\pm$32 ns (Intel GbE chips)
- Some restrictions
  - Packets must be UDP or PTP L2 protocol
  - Minimum UDP packet size is 84 bytes
- Some NICs support timestamping arbitrary packets (e.g., Intel Xeon D embedded X552, or Intel E810-based NICs)

Packet Generators
Rate control

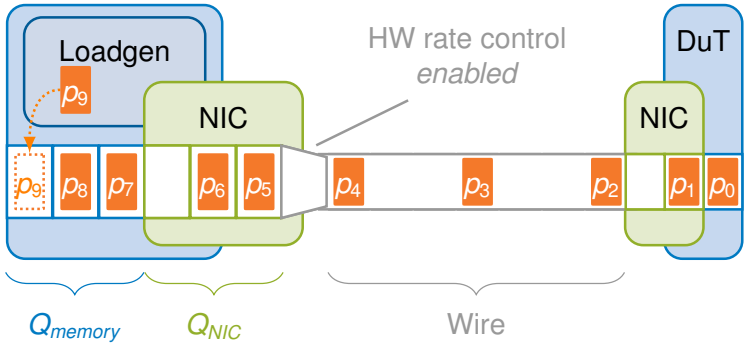
Software rate control in existing packet generators



- Software tries to push single packets to the NIC
- Queues cannot be used, no batch processing
- NICs work with an asynchronous push-pull model
- This can lead to micro-bursts

→ Unreliable, imprecise, and bad performance

Packet Generators
Rate control
Hardware rate control

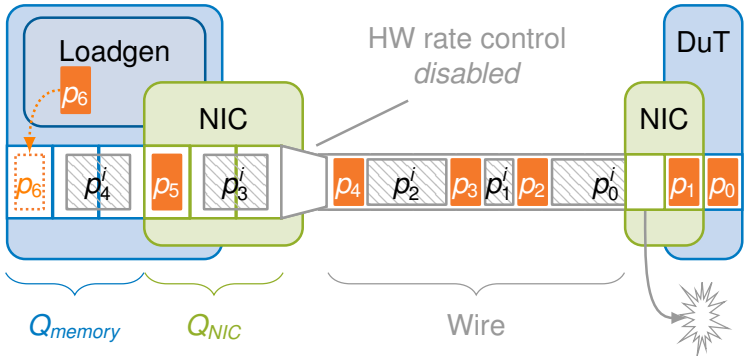


- Modern NICs support rate control in hardware
- Limited to constant bit rate and bursty traffic
- Precision controlled by the hardware

➔ High performance as queues can be used, but inflexible

Packet Generators
Rate control

Software rate control based on invalid packets

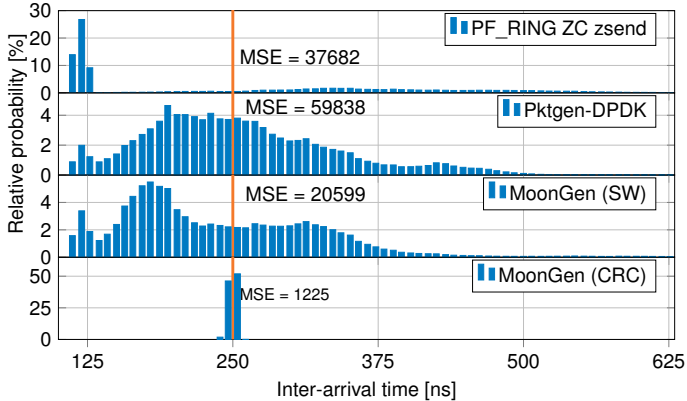


- Fill gaps with invalid packets p^i (e.g. bad CRC)
- NIC in the DuT drops invalid packets without side-effects
- Combines advantages of both approaches
- Precision limited by byte rate (0.8 ns per byte) and minimum packet size (33 byte)

→ High performance & high precision

Packet Generators
Precision of software rate control

4 Mpps with constant bit rate

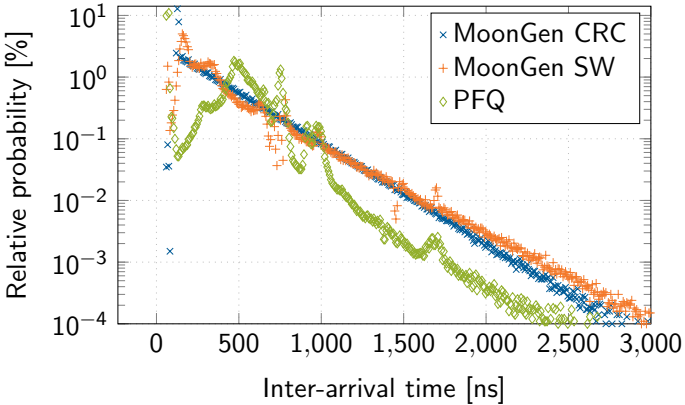


- Histogram of packet inter-arrival times, orange line marks target (200 ns) for CBR traffic
- Bursts with an inter-arrival time of 115 ns (128 Byte packets at 10 Gbit/s) are particularly bad

Precision of software rate control

3 Mpps, Poisson process

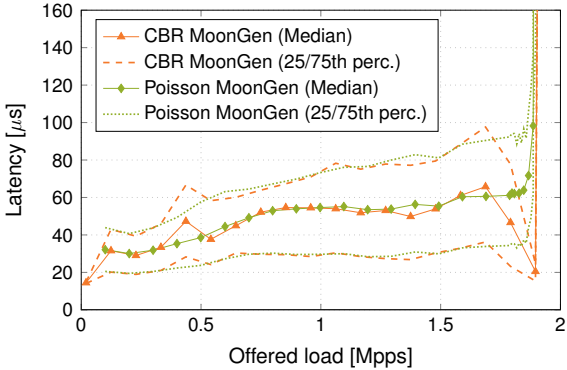
- Poisson process: exponential distribution of inter-arrival times (PDF: $f(x) = \lambda e^{-\lambda x}$)
- Poisson is easier to generate and scale: Poisson processes can be merged (multi-threading)
- Histogram of a poisson process on a logarithmic plot should be a straight line



Packet Generators
Precision of software rate control

Does it matter?

- Device under test: Open vSwitch on Linux, measuring latency
- Compare CBR with Poisson traffic, change only the traffic pattern between tests
- Different response from the device under test when measuring latency



Performance Measurements

Further reading

What this lecture did not cover:

- Virtual machines
- Effects of CPU caches, hyper-threading, energy saving, clock frequency, ...
- Other operating systems (FreeBSD, Windows, ...)
- Traditional socket API
- Hardware latency measurements
- Lots of details (NAPI, interrupts, ...)

Performance Measurements

Performance Measurements

We recommend the following publications if you are interested

[1] P. Emmerich, "Demystifying Network Cards," in 34th Chaos Communication Congress https://media.ccc.de/v/34c3-9159-demystifying_network_cards, Dec. 2017.

[2] P. Emmerich, S. Ellmann, and S. Voit, "Safe and Secure Drivers in High-Level Languages," in 35th Chaos Communication Congress https://media.ccc.de/v/35c3-9670-safe_and_secure_drivers_in_high-level_languages, Dec. 2018.

[3] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A Scriptable High-Speed Packet Generator," in Internet Measurement Conference (IMC) 2015, IRTF Applied Networking Research Prize 2017, Tokyo, Japan, Oct. 2015.

[4] L. Rizzo, "netmap: A Novel Framework for Fast Packet I/O," in USENIX Annual Technical Conference (ATC), 2012.

[5] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, "Comparison of Frameworks for High-Performance Packet IO," in ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2015), 2015.

[6] P. Emmerich, D. Raumer, S. Gallenmüller, F. Wohlfart, and G. Carle, "Throughput and Latency of Virtual Switching with Open vSwitch: A Quantitative Analysis," in Journal of Network and Systems Management, Jul. 2017. DOI: 10.1007/s10922-017-9417-0.

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