Technische Universität München

TUM School of Computation, Information and Technology

Fast and Flexible Software-based Packet Processing Systems

Paul M. Emmerich

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

genehmigten Dissertation.

Vorsitzender: Prof. Dr.-Ing. Pramod Bhatotia

Prüfer der Dissertation:

1. Prof. Dr.-Ing. Georg Carle
2. Prof. Andrew W. Moore, Ph.D.

Die Dissertation wurde am 01.06.2022 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 31.10.2022 angenommen.

Abstract

Packet processing systems like routers or firewalls at the core of Internet infrastructure are currently special-purpose hardware devices. There is an ongoing revolution that is invisible to outside observers: More and more functionality is moving from hardware to software. This yields advantages related to flexibility and costs: Anything can be deployed on inexpensive commodity hardware and adding new functionality merely requires a software update. However, the performance characteristics of software are less well-understood than those of the hardware boxes that are being replaced.

Our thesis is that we can utilize high-level programming languages to build fast and flexible packet processing systems which can replace hardware with software in many scenarios. This dissertation is a walk through the stack of a software packet processing system, investigating performance and flexibility at every layer. We present ixy, our own network driver which we implement in a total of ten different programming languages: C, Rust, Go, C#, Java, OCaml, Haskell, Swift, JavaScript, and Python. We identify key components and parameters for improving throughput and reducing latency: batch processing of 32 packets per batch and DMA ring buffer sizes of 512 are the sweet spot. Our fastest implementation is in C and achieves a packet forwarding rate of 27.4 million minimum-sized packets per second (18.4 Gbit/s) on a single 3.3 GHz CPU core. Rust achieves 98% of C's throughput at the same worst-case latency of 22 µs (median 5 µs) while at the same time guaranteeing memory safety in 87% of the source code. The other languages fare worse and should not be the first choice for building a packet processing system.

Testing software-based packet processing systems is also more difficult compared to pure hardware systems. More performance-affecting effects need to be taken into account and the additional complexity enabled by moving to software means that the testing device and methodology also needs to be more complex. Traditional hardware-based packet generators can also be replaced with pure software variants. We present MoonGen, a software packet generator written in the high-level programming language Lua. Precision is a challenge for software, especially for packet generators attempting to measure latency. MoonGen's timestamping achieves a precision of ±12.8 ns by utilizing hardware features available on commodity NICs, effectively making it a hybrid system combining the best of the hardware and software worlds. A second problem is precisely controlling when a packet is sent. Evaluations using the NetFPGA platform show a mean squared error of 1.2 µs for a hardware-assisted approach and 20.6 µs for a pure software implementation. The best prior software-based packet generator we evaluated only achieved a mean squared error of 59 µs. MoonGen has become the de-facto standard packet generator used in academia for evaluating software-based packet processing systems.

Zusammenfassung

Paketverarbeitungssysteme wie Router oder Firewalls, die das Herzstück der Internet-Infrastruktur bilden, basieren derzeit auf spezialisierten Hardware-Geräten. In einer für Beobachter unsichtbaren Transformation werden immer mehr dieser Funktionen in reine Software verlagert. Daraus ergeben sich Vorteile in Bezug auf Flexibilität und Kosten: Alles kann auf preiswerter Standardhardware bereitgestellt werden, und das Hinzufügen neuer Funktionen erfordert lediglich ein Software-Update. Allerdings sind die Leistungsmerkmale von Software weniger gut erforscht als die der Hardware, die hier ersetzt wird.

Unsere These ist, dass wir mit Hilfe von Hochsprachen schnelle und flexible Paketverarbeitungssysteme bauen können, die in vielen Szenarien Hardware durch Software ersetzen können. Diese Dissertation geht systematisch durch den Stack eines Software-Paketverarbeitungssystems, indem Leistung und Flexibilität auf jeder Ebene untersucht wird. Wir stellen ixy vor, unseren eigenen Netzwerktreiber, den wir in insgesamt zehn verschiedenen Programmiersprachen implementieren: C, Rust, Go, C#, Java, OCaml, Haskell, Swift, JavaScript und Python. Wir identifizieren Schlüsselkomponenten und -parameter zur Verbesserung des Durchsatzes und zur Verringerung der Latenz: Batchverarbeitung von 32 Paketen pro Batch und DMA-Ringpuffergrößen von 512 stellen sich als ideal heraus. Unsere schnellste Implementierung ist in C geschrieben und erreicht auf einem einzelnen 3,3 GHz CPU-Kern eine Paketweiterleitungsrate von 27,4 Millionen Paketen pro Sekunde, bzw. 18,4 Gbit/s bei minimaler (64 Byte) Paketgröße. Rust erreicht 98% des Durchsatzes von C bei der gleichen Worst-Case-Latenz von 22 µs (Median 5 µs) und garantiert gleichzeitig Speichersicherheit in 87% des Quellcodes. Die anderen Sprachen schneiden schlechter ab und sollten nicht die erste Wahl für die Implementierung eines Paketverarbeitungssystems sein.

Auch das Testen softwarebasierter Paketverarbeitungssysteme ist im Vergleich zu reinen Hardwaresystemen komplexer. Es müssen mehr leistungsbeeinflussende Effekte berücksichtigt werden, und die zusätzliche Komplexität, die durch die Umstellung auf Software ermöglicht wird, bedeutet, dass auch das Testgerät und die Methodik komplexer sein müssen. Traditionelle hardwarebasierte Paketgeneratoren können auch durch reine Softwarevarianten ersetzt werden. Wir stellen MoonGen vor, einen Software-Paketgenerator, der in der Hochsprache Lua geschrieben ist. Durch die Verlagerung der Paketerzeugung in eine Skriptsprache auf der Grundlage eines Userspace Netzwerktreibers können wir die Herausforderungen überwinden, mit denen frühere Generationen von softwarebasierten Paketgeneratoren konfrontiert waren. Präzision ist eine solche Herausforderung, insbesondere für Paketgeneratoren die Paketlaufzeiten messen. Das Timestamping von MoonGen erreicht eine Genauigkeit von 12,8 Nanosekunden, indem sie Hardwarefunk-

tionen von handelsüblichen Netzwerkkarten nutzt. MoonGen ist ein hybrides System, das das Beste aus der Hardware- und der Software-Welt vereint. Ein zweites Problem ist die genaue Kontrolle darüber wann ein Paket versendet wird. Auswertungen mit der NetFPGA-Plattform zeigen einen mittleren quadratischen Fehler von 1,2 µs für unseren Hardware-gestützten Ansatz und 20,6 µs für unsere reine Software-Implementierung. Der beste von uns untersuchte softwarebasierte Paketgenerator in der Literatur erreichte nur einen mittleren quadratischen Fehler von 59 µs. MoonGen hat sich zum De-Facto-Standard für die Evaluierung von softwarebasierten Paketverarbeitungssystemen im akademischen Bereich entwickelt.

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Introduction

This dissertation explores and analyses performance behavior due to the shift from special-purpose hardware appliances to software for packet processing devices. We investigate how packet processing can be made cheaper, safer, and more flexible by using software in a domain that was traditionally dominated by hardware.

1.1 Motivation

Packet processing systems are systems handling individual packets on OSI layers 2 and 3, e.g., routers, firewalls, or virtual private network (VPN) gateways. They can also process parts of layer 4 such as ports or header flags. Actions taken by these systems are typically forwarding, filtering, inspecting, or encapsulating individual packets. This puts packet processing systems at the core of networking infrastructure, in contrast to endpoints such as web servers caring about streams of data on higher layers.

Many of these functions were traditionally implemented in special-purpose network appliances – expensive and inflexible black-boxes implementing specific logic functions in hardware. As new protocols are invented and deployed, network operators need to upgrade these boxes. Upgrading means replacing if the implementation is an application-specific integrated circuit that simply does not support the desired new features. Moreover, flexibility in the form of programmability of all network devices network can increase reliability, security, and performance of the network as a whole [49].

There are two approaches to make upgrading functionality easier and cheaper:

1. Make devices programmable via an open interface, e.g., OpenFlow [103] or P4 [21].
2. Run the full processing stack in software on commodity off-the-shelf hardware, e.g., by using netmap [148] or DPDK [96].

Both of these solutions are relatively old in this fast moving field: OpenFlow was published in 2008 [103]. The Click modular router demonstrated increasing flexibility in routers via a pure software implementation in 1999 [92].

This dissertation is concerned with the software approach which was reinvigorated by software frameworks offering fast packet IO such as netmap (2012) and DPDK (2013) [148, 96]. Yet, both these frameworks and the network drivers they are built upon are written in the low-level language C. The same is true for packet processing systems built on top of them (Section 4.3.5). Our thesis is that we can increase the flexibility and safety properties of these systems further without compromising performance by using high-level languages instead.

This dissertation is a practical demonstration of the feasibility of packet processing systems in high-level languages. We build a network driver from scratch in ten different programming languages: C, Rust, Go, C#, Java, OCaml, Haskell, Swift, JavaScript, and Python (Chapter 4). Our Rust implementation achieves 98% of C's performance while guaranteeing memory safety in 87% of the code.

Another problem to tackle when moving from hardware to software systems is benchmarking them properly. Software can exhibit performance characteristics that are unexpected for users only familiar with hardware implementations. Benchmarking methodology differs between software and hardware devices under test, Section 2.3 discusses how many standards are designed with hardware devices in mind. Test equipment such as packet generators are also traditionally built as special-purpose hardware devices with limited flexibility when it comes to supporting new protocols. Software can also help here: In Chapter 5 we discuss how a software-based packet generator running on commodity hardware can be made cheaper and more flexible than a hardware device.

We again take an approach relying on an artifact here: we build the packet generator MoonGen that is implemented in the high-level programming language Lua (Chapter 5). There are several challenges to overcome: Existing hardware devices are very good at being fast and precise, tasks that can be hard for pure software solutions. For example, test equipment needs to be able to measure latencies with a high precision, an easy task for a dedicated hardware timestamping device. These problems get worse as you move further away from the hardware through more and more abstraction layers. For example, high-level languages add unpredictable sources of latency such as garbage collectors and just-in-time compilers into the mix that needs to be taken into account. MoonGen uses hardware features found on commodity NICs to overcome these limitations, achieving a timestamping precision of ± 12.8 ns and a mean squared error of 1.2 µs for controlling when a packet is transmitted (Chapter 6).

1.2 Research Questions

This dissertation addresses the following main research questions.

Q1 What makes software-based packet processing fast?
We need to gain a deep understanding about the basic principles of why some software-based systems are fast and others are slow. Network devices in software are not new per se, but fast network devices that only use software are a recent trend. We examine which foundations such software must be built upon to be fast by looking at the deepest layer in a software stack: the device drivers. Our approach to answering this is by building a new driver and packet IO framework from scratch to evaluate the effects of individual optimizations in isolation (Chapter 3).

Q2 Can high-level languages be used in software-based network devices?
C is the go-to language for code that is considered low-level such as the applications investigated here. We investigate if and how we can use high-level languages and look at the trade-offs incurred by them. High-level languages can come with overheads for performance (e.g., bound checks on memory access) and latency (e.g., garbage collection or just-in-time compilation). But their use can allow for a higher flexibility and improved safety. Do all high-level languages suffer from these overheads? How can these trade-offs be quantified when they are present? We approach this question by implementing drivers in different high-level languages and benchmark them (Chapter 4).

Q3 How can modern network devices be benchmarked?
Today's network devices require a more flexible test approach than using a hardware device for benchmarking. For benchmarking we focus on the metric of packets per second (pps) instead of bits per second as per-packet costs are the primary bottleneck (Section 2.2). Especially software-based devices pose a challenge: they support new protocols and can exhibit performance characteristics only visible under test conditions that are complex to produce. We apply the learnings about performance and flexibility of high-level languages to create MoonGen (Chapter 5), a novel benchmarking tool that is up to these challenges.

Q4 Can software be sufficiently precise to replace hardware in all scenarios?
Replacing hardware with software entails the risk of losing the features provided by hardware. In particular, hardware can provide high precision. This leads us to consider the following questions: In which scenarios is a software-only solution sufficient? What are the trade-offs and how can they be quantified? We quantify the precision of MoonGen by testing it with the hardware packet generator OSNT [7] running on the NetFPGA platform (Chapter 6).

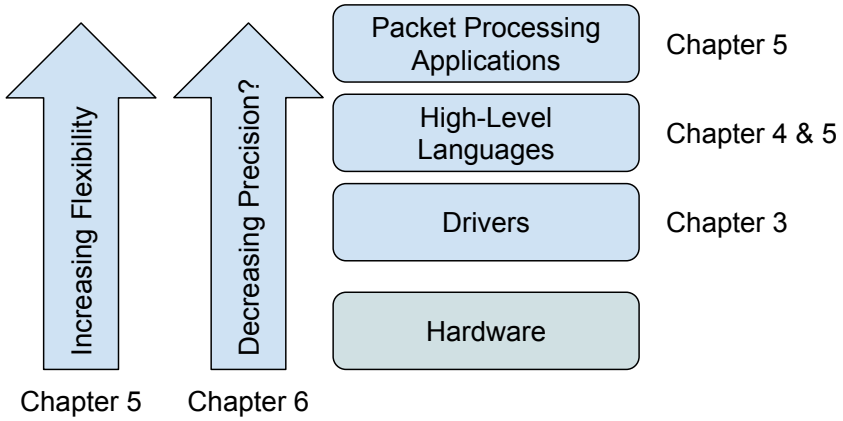


Figure 1.1: Structure of this dissertation

1.3 Structure of this Dissertation

This dissertation is structured in two main parts. We first look at the lowest, and most crucial, part of the software stack: the drivers. Chapter 3 investigates how modern user space drivers work in general by presenting the reference implementation ixy in C. We discuss the architecture and design choices, their impact on performance and the implementation in deep detail. Chapter 4 builds on this by adding high-level languages to the mix: How can they help us to achieve our goals? What are their performance impacts?

The second part looks at higher layers: an application in a high-level language running on such drivers. Chapter 5 presents the packet generator MoonGen written in the high-level language Lua. We discuss trade-offs between flexibility and performance. Finally we evaluate trade-offs between precision requirements and flexibility by evaluating the precision of MoonGen. Which features that traditionally required a hardware implementation can be done in software with sufficient precision and accuracy?

Figure 1.1 visualizes the building blocks of modern packet processing applications to the right. Chapters 3 to 5 go through the stack starting right above the hardware and ending at an application written in a high-level language. Going up this stack increases flexibility, Chapter 5 discusses how. At the same time precision decreases, Chapter 6 quantifies by how much and what trade-offs can be made to keep it precise enough for even a high-precision test tool.

1.4 Key Contributions

This dissertation contributes several important artifacts to the networking research community as well as several key insights into the behavior of complex packet processing systems.

1.4.1 The ixy Network Driver

ixy is a high-speed user space network driver targeting the 10 Gbit/s Intel ixgbe family of network cards (Chapter 3). The driver is stripped down to the bare minimum required for high-speed packet processing in a modern network application. Its barebone nature allows to evaluate and benchmark individual optimizations or features in isolation, helping us to understand what makes software-based packet processing fast. For example, we use these drivers to quantify the effects of interrupts (Section 3.5.5), IOMMU isolation and page size optimizations (Section 3.5.9), and cache prefetching (Section 3.5.4) in a minimal setup. ixy can forward 27.4 million minimum-sized packets per second (18.4 Gbit/s) on a single 3.3 GHz CPU core (Section 3.5).

But it is more than a building block for research, its simple and well-documented code base is also an asset for education to aid understanding network drivers at the lowest level. For this purpose it has been made available under a free and open source license [44].

1.4.2 Network Drivers in High-Level Languages

To understand how high-level languages can be utilized in software-based network devices we re-implemented the whole ixy driver in the high-level languages C, Rust, Go, C#, Java, OCaml, Haskell, Swift, JavaScript, and Python (Chapter 4). All implementations in high-level languages use the ixy driver, which is written in C, as a template. They implement the same architecture and same feature set, allowing for a comparison of different languages.

These drivers, collectively called the ixy-language drivers, were implemented by a team of students (advised by the author of this dissertation) with experience in the respective programming language to ensure idiomatic implementations in the different languages. We use these drivers to quantify the trade-offs incurred by different language features. For example, we can measure the impact of garbage collection on the performance and latency to quantify the trade-off of traditional memory safe languages vs. performance (Section 4.6.2). Our Rust implementation achieves 98% of C's throughput at the same worst-case latency of 22 µs (median 5 µs) while at the same time guaranteeing memory safety in 87% of the source code (Sections 4.7, 4.4.3).

1.4.3 The MoonGen Packet Generator

MoonGen is a corner stone of this dissertation, it is a fully scriptable packet generator tuned to be as flexible as possible by replacing hardware with software and the high-level language Lua (Chapter 5). It executes custom user-provided Lua script code for every single packet it sends out and the whole configuration and test definition is done via user-provided scripts (Section 5.4). It can be seen as a framework for building packet generators and tests tailored to a specific system under test. Despite this flexibility it achieves 14.88 Mpps on a single 1.5 GHz CPU core (Section 5.5.2).

We combine the advantages of software and hardware packet generators while avoiding their respective disadvantages. Our MoonGen implementation introduces a novel software method for precise control over inter-packet gaps, achieving a mean squared error of 1.2 µs, the previous state of the art was 59 µs for software-based packet generators (Section 6.6.4). Precise (±12.8 ns) time-stamping is achieved by exploiting a hardware feature that is commonly found on commodity server hardware: support for the PTP protocol (Section 5.6). It is not only relevant to answer the question how to benchmark modern network devices, it also looks at using high-level languages for domains traditionally dominated by hardware and/or low-level languages.

MoonGen is an especially impactful artifact developed for this dissertation, we use it to evaluate and compare all of our other implementations. It also became the de-facto standard benchmarking tool for network performance researchers and has been used by several other high-impact publications, e.g., [152, 93, 34, 168].

1.4.4 Hardware-based Precision Evaluation of Packet Generators

Understanding the limitations of your testing tools is important in order to design experiments that produce valid results. We present a detailed comparison of the precision of various packet generators commonly used in academia (Chapter 6), uncovering shortcomings of commonly used tools that threaten the validity of experiments.

Our experiments use the NetFPGA platform to perform precise and accurate measurements of network tools, validating MoonGen against a hardware packet generator. These experiments allow us to assess and quantify in which scenarios software implementations lag behind hardware solutions. Insights obtained from these experiments allow us to quantify when a software implementation can be precise enough for a given requirement. Our findings indicate that MoonGen is precise enough to characterize latency behavior of both software and hardware-based packet processing systems.

Chapter 2

Background

There is an ongoing move from special-purpose hardware to software solutions for many key components in the Internet infrastructure. Appliances providing network functions such as routing, switching, or firewalling are traditionally implemented as expensive proprietary black-boxes. This approach comes with an inherent low flexibility: for example, upgrading a network from IPv4 to IPv6 requires replacing routers that were built without IPv6 support in mind. Compare this to a modern software router: it can be upgraded to speak new protocols above layer 2 without replacing hardware.

Dedicated hardware comes with hard performance guarantees: there are well-defined limits such as sizes of look-up tables or the line rate of a network interface adapter. One can expect the devices to perform in a predictable manner. Many of these nice properties get lost when moving to software. A software implementation can be affected by many factors on all involved layers from the available hardware resources to peculiarities of a specific programming languages runtime such as just-in-time compilation or garbage collection. This behavior also poses challenges for testing methodology and software. Many tools and procedures for testing were developed with hardware as systems under test in mind.

2.1 Hardware Architecture of Software-based Packet Processing System

Software needs hardware to run on. In this dissertation we build and evaluate our software on Intel network cards and CPUs because Intel provides extensive documentation on internal details, enabling development of a new driver from scratch. Intel's detailed

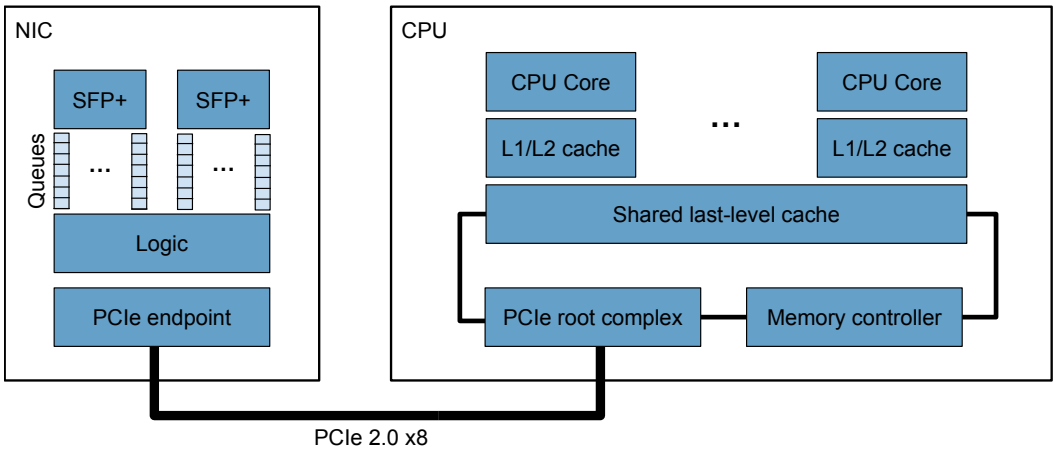


Figure 2.1: Overview over hardware architecture of a representative server system used in this dissertation

datasheet for the 82599 networking chip [73] has been an invaluable resource in building both the ixy network drivers and the MoonGen packet generator.

Figure 2.1 shows the important components of one of the server systems used in this work. The CPU is an Intel Xeon E5-2620 v3 (Haswell EP) clocked at 2.4 GHz with 6 cores, the NIC is an Intel X520-T2 featuring an 82599 network interface controller chip and two SFP+ (small form-factor pluggable) ports that can be equipped with different transceivers supporting up to 10 Gbit/s. A PCIe 2.0 x8 link is used to connect the NIC to the controller which can seem somewhat antiquated, but it supports a bandwidth of 32 Gbit/s in each direction, enough for the two 10 Gbit/s Ethernet ports.

The key component of the NIC used here are the queues. Each physical port features multiple independent receive and transmit queues, this simplifies scaling of multi-flow traffic. The driver can configure filters or hashing over packet headers to distribute incoming traffic into different queues. Outgoing traffic from multiple queues is mixed together in hardware by the NIC. Each queue comes with its own DMA memory area, so typically there is one pair of queues for each CPU core to achieve multi-core scaling. [73]

The key component of the modern CPU is that it has an integrated PCIe root complex that can directly access the system's last-level cache, on the CPU depicted here via a ring bus [80]. This implies that the main memory is ideally never involved in processing a packet as a typical server CPU features a last-level cache in the size of tens of megabytes (15 MiB here) – enough to fit a large burst of packets. However, in a modern CPU even the last-level cache can be quite *far away* from the cores doing the actual processing from a latency perspective: On Haswell EP the access time from a CPU core to the L3 cache is 53 clock cycles [109]. Even the L1 and L2 caches have an access time of 4 and

11 cycles respectively [80]. Moreover, the cache is organized *cache lines* of size 64 byte that are always cached together. That means attempting to access a single byte that is already in L3 cache but not in the core's L1 or L2 cache will incur the 53 cycle penalty, but subsequent accesses to adjacent bytes only need to wait for 4 cycles.

Note that custom processing logic in a packet processing system may need to go to main memory, e.g., to consult large lookup tables or other data structures. All of our use cases typically fit entirely into the cache, for example, the entire IPv4 routing table used for the Internet fits in 2.4 MB of memory (IPv6: 1.4 MB) when using a data structure optimized for both lookup performance and size [10]. Any optimizations to larger data structures that may be required are orthogonal to the work presented here.

Relevant details of the test systems used for the individual experiments are given in the sections describing the respective test setups as multiple different systems were used throughout this dissertation. The system depicted above is merely a representative example of a modern server systems. Details can vary between different systems, but the core components explained above generalize well. Multiple queues on NICs are standard across high speed NICs: splitting a port into multiple independent queues is a de-facto standard abstraction layer for high-speed network drivers, e.g., see the interfaces of netmap and DPDK [96, 148]. Modern general-purpose multicore CPUs also all look alike when it comes to the cache hierarchy and embedded memory and PCIe controllers.

2.2 Important Performance Numbers

We are targeting 10 Gbit/s Ethernet connections in this work, but raw throughput as measured in bits per second is rarely a useful metric for packet processing systems. The systems we are building are mainly concerned about packet headers, not their payload. Even systems that process payloads, such as VPN gateways without hardware offloading for cryptography, are dominated by per-packet costs [140].

So how many packets per second fit through a 10 Gbit/s Ethernet connection if they are all as small as possible? **14.88 million packets per second (Mpps)**. This is an important number and it will show up again and again throughout this dissertation. You can derive this number from the minimum Ethernet frame size of 64 bytes. On top of this you have to take into account 8 bytes of overhead when packaging an Ethernet frame into an Ethernet packet: 7 bytes preamble and a 1 byte start-of-frame delimiter. In addition, two Ethernet packets are separated by 12 bytes of inter-packet gap for an effective total minimum size of 84 B a packet on the wire [67]. Dividing 10 Gbit/s by 84 B yields the number of 14,880,952 packets per second.

A related measure is the time it takes to serialize a packet onto the wire: 67.2 ns as a 10 Gbit/s link transmits a byte in 0.8 ns. This is also the time our system can spend to process a packet *on average*. Throughout this dissertation we use CPUs with clock frequencies of 2.4 GHz to 3.3 GHz, so we only have **161 to 222 clock cycles** available for each packet.

This seems like an impossible task at first glance when comparing this to cache latencies given in the previous section. It takes 53 clock cycles to get the packet from the L3 cache to the CPU core, so a third to a quarter of the available time is spent just waiting on data. The key insight here is that our performance target is only an average. Packets are queued (we find a buffer size of 512 packets ideal, see Section 3.5.7) and processed in batches. We can amortize costs such as cache load times across all packets within a batch by prefetching packet data into caches closer to the CPU cores. See Sections 3.5.3 and 3.5.4 for details.

2.2.1 Performance Targets

One of the applications we are building is MoonGen, a benchmarking tool. It must be able to handle the absolute worst case—small packets in a single flow at the highest possible speed—in order to characterize the behavior of other systems under the worst case. This means we must be able to achieve 14.88 Mpps in order to handle 10 Gbit/s Ethernet.

Moreover, this speed must be achieved while using only a single CPU core: Packets within a single flow cannot be effectively split across multiple CPU cores because of dependencies between packets for processing logic and ordering requirements. Memory synchronization between multiple cores is too slow [109] to make data sharing between multiple cores a feasible option for a single flow.

Scaling to multiple CPU cores relies on having multiple independent flows in the packets that are being processed. As explained in Section 2.1, a common feature supported in hardware by high-speed NICs is distributing incoming packets to different queues based on a hash across customizable header fields. This effectively balances flows across queues in hardware and each queue can be handled by a dedicated CPU core. We lose the control over the order of packets between different flows (inter-flow ordering) with this approach, but this is not an important property as flows are independent. But the more important property of packet order within a flow (intra-flow ordering) is preserved since each flow is handled by the same CPU core. Hence, most experiments in this dissertation will focus on achieving **14.88 Mpps on a single core** as this is the main property we focus upon.

2.3 Benchmarking methodology and packet generators

RFC 2544 [25] is a standard specifying a suite of tests to benchmark networking devices. Vendors of hardware networking devices commonly offer performance evaluations following this standard. Yet, with a publication date of March 1999 the standard is now well over 20 years old – an eternity in this fast moving field.

One particular problem with this standard is the way latency measurements are specified. It calls for measuring the latency of a single packet after 60 seconds of test time and then repeating that 20 times and reporting the average. While this methodology is perfectly reasonable for a completely predictable device, it is not sufficient for a software device. For example, a software router that uses a high-level language featuring a garbage collector might suffer from unpredictable and rare pause times. Many more measurements are required to find these.

The Linux Foundation project "Open Platform for NFV" features the sub-project VSperf which takes a look at performance characterization of virtual switches [121]. They published a summary of their methodology as RFC 8204 [158] in 2017 which addresses many of the problems that showed up when considering software instead of hardware boxes as systems under test. For example, they explicitly talk about latency distributions and outliers instead of relying on a simplistic metric like just the average. However, the RFC is intentionally very vague about the exact measurement parameters and report formats to use compared to the old RFC 2544.

For this dissertation, we use these standards as guides for making sure our tests follow the best practice for test setups and consistent terminology. Strictly following the prescribed methodology is too limiting: RFC 2544 is too old to be applicable and RFC 8204 is too vague in parameters. We pick ideas and tests that make sense in context and add our own. In particular, we conduct many white-box tests whereas standards only describe black-box tests.

It should be noted that the development of RFC 8204 overlapped with the work on this dissertation. In fact, the packet generator MoonGen developed for this dissertation is used in the reference implementation of the VSperf project and the author of this dissertation attended multiple IETF meetings discussing this new RFC.

2.4 Test Setups Used in this Dissertation

Table 2.1 gives the relevant hardware information about the primary test systems used throughout this dissertation. As this work was done over the course of several years

	CPU	Frequency	Cores	NICs
Server 1	Intel Xeon E5-2620 v3	2.4 GHz	6	Intel 82599, X540, X550, XL710
Server 2	2x Intel Xeon E5-2630 v4	2.2 GHz	20	Intel 82599, X540, X550, XL710
Server 3	Intel Xeon E3-1230 v2	3.3 GHz	4	Intel 82599
Server 4	Intel Xeon E3-1230	3.2 GHz	4	Intel 82599
Server 5	Intel Xeon D-1537	2.3 GHz	8	Intel X552 (SoC)

Table 2.1: Server systems used for experiments throughout this dissertation

(experiments conducted between 2014 and 2020) some details (e.g., swapping NICs, cabling, software upgrades) vary between the individual setups. Details about software and other hardware, where relevant, is given in the description of the experiments using the setups throughout this dissertation.

Server 1 and Server 2 are directly connected with 10 Gbit/s and 40 Gbit/s direct attach copper cabling for the (Q)SFP+ NICs Intel 82599 and XL710, and Cat 5e cables for the X540 and X550. These serve as the main development servers on which the bulk of this work was done. Most experiments in this dissertation were conducted on Server 1, and it is also the one used as an example in Section 2.1. Server 2 is used for experiments requiring a NUMA architecture and as a packet generator running MoonGen (Chapter 5) for experiments on Server 1.

Server 3 and Server 4 are also directly connected to each other, in this case with single mode fiber cabling. Server 3 serves as system under test, and server 4 as packet generator. Server 5 is connected via a passive tap device (fiber optic splitter) to intercept traffic between servers 3 and 4, see Figure 2.2.

2.4.1 Latency Measurement Setups

MoonGen's latency measurements are restricted to sample the latency of random packets in the stream, it cannot measure the latency of every single packet (Section 5.6.4). This is more than sufficient to match requirements for common benchmarking standards (Section 2.3) and also yields enough data for almost all experiments devised for this disseration. Yet, there is one experiment in this work for which this was not sufficient: In Section 4.7 we take a close look into the worst-case behavior and jitter introduced by just-in-time compilation and garbage collection in high-level languages, we need to capture more packets than a simple test setup supports to reliably catch rarely occurring events.

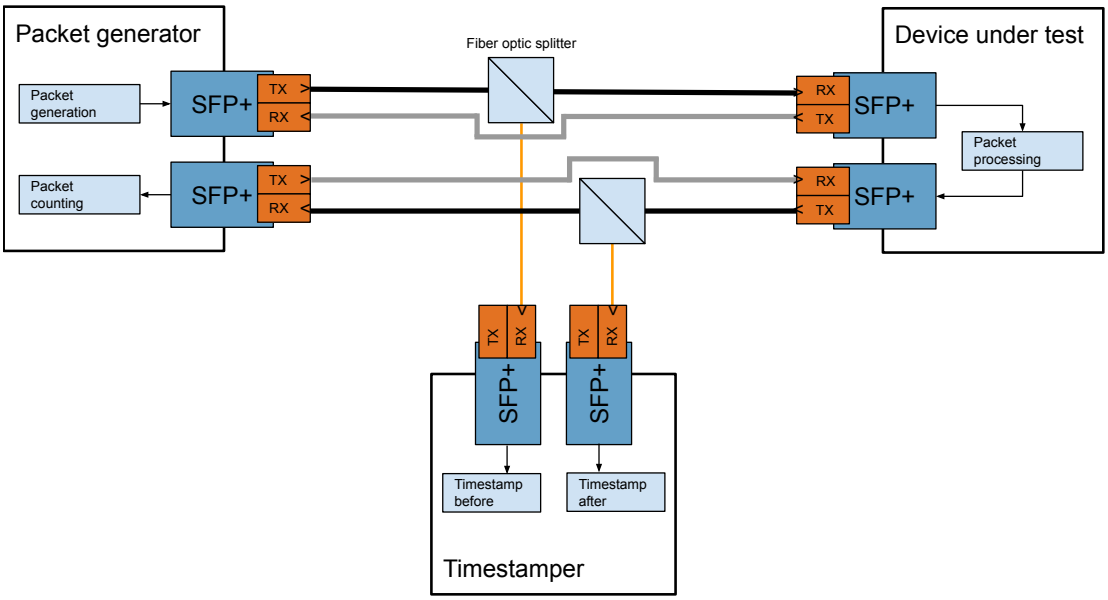


Figure 2.2: Wiring with fiber optic splitters to measure latencies via a third server.

Figure 2.2 shows a setup that allows us to intercept packets in one direction before and after the device under test by inserting fiber optic splitters into the setup. The packet generator inserts a sequence number into each packet which is used by the dedicated timestamping server to correlate the two copies of the received packet. This timestamper needs hardware that is able to capture the timestamps of all received packets. Doing this typically requires expensive dedicated timestamping hardware [4], it is a rare feature on inexpensive commodity NICs. We use an Intel X552 NIC which is embedded in the Xeon D-1537 system on chip CPU in Server 5. This is the only commodity NIC with an SFP+ port that features this timestamping capability that was available. On the software side we run MoonSniff, based on MoonGen (Section 5), which achieves an accuracy of 20 ns for latency measurements [5, 4].

2.4.2 System Configuration

We disabled the CPU's automatic frequency adjustment features to get reproducible results. All processes of the software under test were always explicitly pinned to CPU cores to avoid noise due to core migration. Power saving features of the systems were also disabled to reduce noise.

2.5 User Space Packet Processing

This section is based on joint work by Paul Emmerich, Maximilian Pudelko, Simon Bauer, Stefan Huber, Thomas Zwickl, and Georg Carle [44] and joint work by Paul Emmerich, Simon Ellmann, Fabian Bonk, Alex Egger, Esaú García Sánchez-Torija, Thomas Günzel, Sebastian Di Luzio, Alexandru Obada, Maximilian Stadlmeier, Sebastian Voit, and Georg Carle [43].

Traditional software network applications build on top of the socket layer offered by the operating system. This is a flexible solution: the socket API is a stable and battle-tested interface that is available to virtually all programming languages. The problem is that it is by far too slow to be economically viable as a platform to replace existing hardware implementations of network functions.

Network functions have therefore traditionally been built directly into the kernel of operating systems, yielding a performance advantage of about one order of magnitude. Two examples utilizing kernel components are Open vSwitch [132], a virtual switch implementing OpenFlow, and the Click modular router [92], a platform to build network functions via user-provided modules. But writing kernel code is a cumbersome process with slow turn-around times. The environment restricts your choice of programming language: usually only C is possible, sometimes C++, and dynamic or scripting languages are completely missing. A crash of the application can take the whole operating system with it, posing a risk for safety and the lack of isolation mechanisms typically offered by operating systems is a potential security problem. Yet, for many network functions running in the kernel is still the state of the art, for example, the Linux routing and firewalling subsystems are widely used.

The solution is, paradoxically, a move back to user space. Slowness is not an inherent property of a user space application, the problem is only moving data between the application and the hardware. In other words: a new IO interface can fix the problems without having to impose the restrictions of the kernel environment on the application. Over the past decade a large variety of solutions for this have sprung up and network functions with high performance requirements have moved back from the kernel to user space.

This move happened in two steps: First, frameworks like PF_RING DNA (2010) [114], PSIO (2010) [62, 63], netmap (2012) [148], and PFQ (2012) [18] provided a fast-path to the network driver from user space applications. They speed up packet IO by providing a kernel module that maps DMA buffers into a user space application. These frameworks are not user space drivers: all of them rely on a driver running in the kernel, some requiring minor driver modifications. Using these frameworks meant increased performance but often also a loss of access to hardware functionality that was available in the kernel. These APIs all have poor support for hardware offloading features found

on modern network cards. The next step were user space drivers: DPDK [96] (open sourced in 2013) and Snabb [99] (2012) move the entire driver logic into a user space process by mapping PCIe resources and DMA buffers into a user space library, running the driver in the same process as the application via the `uio` or `vfio` Linux subsystems.

An example for this move from a kernel component to something using a user space driver is the Click modular router [92]: It started out as a kernel extension in 1999 and was later sped up with a netmap interface in 2012 as a demonstration of netmap itself [148]. Finally, a DPDK backend was added in 2015 to increase performance even further [15]. Similar migration paths can also be found in other open source projects: Open vSwitch comes with a kernel module and had plans to add both netmap and DPDK [132], the DPDK backend was merged in 2014, the netmap version never left the proof of concept stage [119]. The firewall pfSense [134] started out by using kernel components, then experimented with both netmap and DPDK in 2015 and finally chose DPDK [84] as the path forward.

Network functions performing this move inherited a bias towards paradigms found in the kernel. For example, we show in Section 4.3.5 that there is a bias towards the programming language C in low-level network applications compared to performance-critical network applications on higher layers. Also, the user space drivers are directly derived from the in-kernel version and hence written in C in DPDK. There is room for improvement: we can still increase flexibility for network functions, for example by using more modern programming languages and paradigms as we demonstrate in Chapter 4.

2.6 Network Function Virtualization

Network function virtualization is an umbrella term for all kinds of different network hardware being softwarized. Examples include firewalls, VPN gateways, intrusion detection systems, routers and even seemingly simple devices like switches. The key point here is that the actual function is decoupled from the hardware: A traditional deployment of network functions is not only inflexible when it comes to adding new functionality to existing hardware, but also to the physical location of the devices. A virtualized network can deploy a network function on any server. For example, a mobile virtual network provider can deploy a software network function at the edge in the network of the infrastructure provider without placing their own hardware at every location. [106]

Running the network function in software is a prerequisite to achieve the goals of network function virtualization. Deployment can work via traditional virtualization by deploying a virtual machine image, via containers or via plugins for specialized network applications (e.g., Click [92] or Snabb [99] modules). Our thesis is concerned with the

foundations for building such applications in a highly performant manner, not with the higher layers of the actual application logic. We care about getting network packets to (and from) a NFV application in the fastest possible way while avoiding to impose restrictions on the actual applications.

2.7 Evolution of Networking Hardware

Our thesis is that moving networking processing logic from dedicated hardware to general-purpose CPUs is the way to go in the future. Moving functionality from dedicated hardware into general-purpose CPUs is nothing new. For example, floating point calculations were once the domain of dedicated co-processors such as the Intel 8087 math coprocessor announced in 1980 [126]. Later, this functionality became a standard part of the general-purpose Intel 80486 CPU in 1989 [78]. Moving math functionality is relatively straight-forward because of well-defined abstraction layers in compilers and math libraries such that the transition is mostly seamless for users of the functions.

Some functionality used for software-based packet processing devices is supported by dedicated processing units on the CPUs. For example, the AES-NI instruction set to speed up encryption with AES was added to Intel CPUs in 2008 [74]. It can be used to speed up VPN gateways using AES encryption [94]. Intel's 82599 NIC (2009, 10 Gbit/s) also offers AES offloading capabilities for IPsec on the NIC [73], this functionality was removed from later Intel NICs, e.g., from the XL710 NIC (2014, 40 Gbit/s) [77].

Integrating high-speed network connectivity directly on the CPU is also common. For example, both Intel's Xeon D CPUs (2015) [79] and AMD's Ryzen Embedded series (2018) [138] feature multiple 10 Gbit/s network ports as part of the CPU. However, these NICs are internally still attached via PCIe and not truly embedded in the core, i.e., they are part of a system on chip design. Our driver ixy (Chapter 3) supports the NICs found on Intel Xeon D CPUs as they are a variant of the aforementioned Intel 82599 chip.

Software development drives hardware development and vice versa in an endless *virtuous cycle*. The work done for this dissertation (started in 2014) was during a golden age for packet processing in software: netmap (2012) [148] and DPDK (2013) [96] unlocked high-speed paths to the hardware, allowing researchers and engineers to achieve unprecedented performance in software. The unlocked flexibility enabled innovation in packet processing devices, see Section 2.6. Ever growing demands for performance cause engineers and researchers to look for faster hardware to run their software. For example, the number of hardware queues available on NICs grew from 64 to 768 on Intel NICs between 2009 (Intel 82599) and 2014 (Intel XL710) [77, 73]. However, as

demand for bandwidth grows engineers and researchers are again looking to offload compute-intensive operations to the NICs, driving their development further. As offloading operations grow more complex a whole new world is opening up: running software directly on the NICs, driving software development.

2.7.1 Another Future: Offloading More to the NIC

This dissertation is focused upon 10 Gbit/s networks with some experiments done on 40 Gbit/s connections. However, the increase in raw speed of network links has outpaced the development of processing power of CPUs [169]. So researchers are currently looking into approaches to offload more work onto the NICs to cope with challenges for 100 Gbit/s and 400 Gbit/s systems. Offloading is the exact opposite of what we are advocating here.

Putting more logic into the NIC to cope with higher network speeds is not a new trend. Myricom published a NIC based on their LANai chip which features a general-purpose CPU in 1995 to handle 1 Gbit/s network connections in high-performance computing applications [156].

A more recent example is Corundum (2020), an open source 100 Gbit/s NIC that can be used as a platform for experimental implementations of offloading features and new protocols [48]. One example for an offloading opportunity is the transport layer, NanoTransport demonstrates handling 380[1] million requests per second on the transport layer in a hardware implementation [8].

Beside offloading to dedicated hardware, there is a second interesting approach: bringing the CPU and NIC closer together. One way of achieving this is by integrating the NIC tighter with the CPU and removing the PCIe bottleneck as PCIe link speeds have increased at a slower rate than network link speeds and CPU processing speeds [169]. Lightning NIC is one such proposal which would give a NIC direct access to a core's registers [66]. This not only circumvents the PCIe link, but it also overcomes the latency associated with loading data from the L3 cache into the CPU core (Section 2.1).

A more practical (commercial hardware exists) approach are smart NICs with general-purpose CPUs to bring the CPU closer to the NIC instead of the other way around. Mellanox BlueField NICs feature an entire general-purpose ARM64 system running Linux [104]. These NICs run DPDK internally, which makes porting existing DPDK

[1] In a simulation assuming they can achieve 3.2 GHz for their chip if they were to build an application-specific integrated circuit.

applications straightforward. Optimizing hardware in this way is orthogonal to our work: Lessons learned from this dissertation apply equally to software running on such systems.

Part II

Fast and Flexible User Space Packet Processing

Chapter 3

Fast User Space Network Drivers

At the core of a modern software-based packet processing system lies a fast specialized driver. The default drivers of operating systems can't cope with the requirements of such systems, so several specialized drivers were developed. Commonly used drivers such those found in DPDK, are complex beasts that can be hard to understand. Yet it is important to understand your building blocks, especially those that are your foundations for achieving a good performance.

Our quest to fully understand the software stack of an application leads us right to these foundations. So we start the investigation in this dissertation by writing our own custom network driver. Having a custom network driver allows us to truly understand what it does and how it can achieve a high performance. In this chapter, we take a deep dive into the gory details of high-speed drivers.

The goal for this chapter is to answer our research question Q1: What makes software-based packet processing fast? We need to gain an understanding of the basic principles before we can walk up the stack to higher layers. We design an architecture for a driver and discuss different trade-offs for performance.

The remainder of this chapter is based on our publication about user space network drivers which is joint work by Paul Emmerich, Maximilian Pudelko, Simon Bauer, Stefan Huber, Thomas Zwickl, and Georg Carle [44]. A full account of the author's contributions is given in Section 3.7.

3.1 Introduction and Motivation

Developers and researchers often treat user space drivers as black-boxes that magically increase speed. Abstractions hiding driver details from developers are an advantage: they remove a burden from the developer. However, all abstractions are leaky, especially when performance-critical code such as high-speed networking applications are involved. We therefore believe it is crucial to have at least some insights into the inner workings of drivers when developing high-speed networking applications.

We present ixy, a user space driver and packet IO framework that is architecturally similar to DPDK [96] and Snabb [99]. Both use full user space drivers, unlike netmap [148], PF_RING [115], PFQ [18], or similar frameworks that rely on a kernel driver. ixy is designed for educational use only, i.e., you are meant to use it to understand how user space packet frameworks and drivers work, not to use it in a production environment. Our whole architecture, described in Section 3.3, aims at simplicity and is trimmed down to the bare minimum. We currently support the Intel ixgbe family of NICs (cf. Section 3.4). A packet forwarding application is less than 1,000 lines of C code including the whole network driver, the implementation is discussed in Section 3.4.

It is possible to read and understand drivers found in other frameworks, but ixy's driver is at least an order of magnitude simpler than other implementations. For example, DPDK's implementation of the ixgbe driver needs 5,400 lines of code just to handle receiving and sending packets in a highly optimized way. They implement the same functionality multiple times, providing optimized code paths based on available vector instruction sets and enabled hardware offloading features. ixy's receive and transmit path for the same driver is only 127 lines of code. The trade-off is that our implementation is slower (no CPU-specific optimization) and we support almost no hardware offloading features.

It is not our goal to support every conceivable scenario, hardware feature, or optimization. We aim to provide a platform for experimentation with driver-level features or optimizations. ixy is available under the BSD license for this purpose [2].

3.2 Background and Related Work

A multitude of packet IO frameworks have been built over the past years, each focusing on different aspects with different trade-offs. They can be broadly categorized into two categories: those relying on a driver running in the kernel (Figure 3.1) and those that re-implement the whole driver in user space (Figure 3.2).

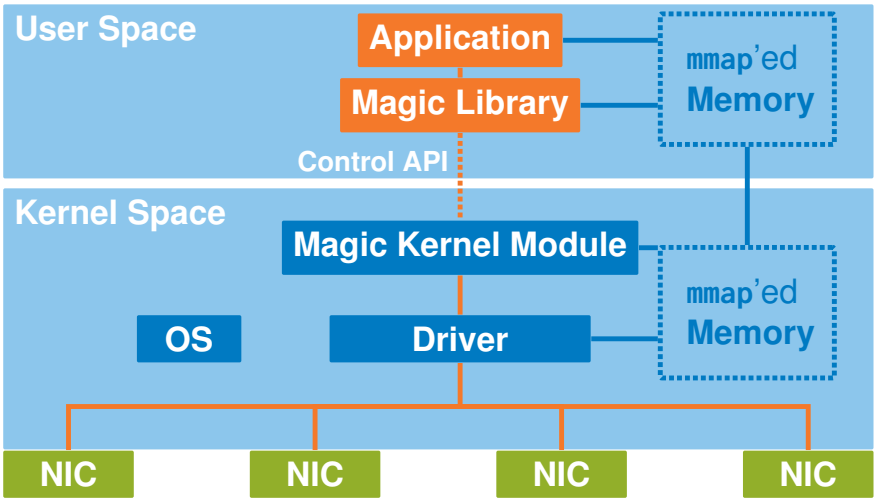


Figure 3.1: Architecture of user space packet processing frameworks using an in-kernel driver, e.g., netmap, PF_RING ZC, or PFQ.

Examples for the former category are netmap [148], PF_RING ZC [115], PFQ [18], and OpenOnload [154]. They all use the default driver (sometimes with small custom patches) and an additional kernel component that provides a fast interface based on memory mapping for the user space application. Packet IO is still handled by the kernel driver here, but the driver is attached to the application directly instead of the kernel datapath, see Figure 3.1. This has the advantage that integrating existing kernel components or forwarding packets to the default network stack is feasible with these frameworks. By default, these applications still provide an application with exclusive access to the NIC. Parts of the NIC can often still be controlled with standard tools like `ethtool` to configure packet filtering or queue sizes. However, hardware features are often poorly supported, e.g., netmap lacks support for most offloading features [50].

Note that none of these two advantages is superior to the other, they are simply different approaches for a similar problem. Each solution comes with unique advantages and disadvantages depending on the exact use case.

netmap [148] and XDP [82] are good examples of integrating kernel components with specialized applications. netmap (a standard component in FreeBSD and also available on Linux) offers interfaces to pass packets between the kernel network stack and a user space app, it can even make use of the kernel's TCP/IP stack with StackMap [167]. Further, netmap supports using a NIC with both netmap and the kernel simultaneously by using hardware filters to steer packets to receive queues either managed by netmap or the kernel [17]. XDP is technically not a user space framework: the code is compiled to eBPF which is run by a JIT in the kernel, this restricts the choice of programming

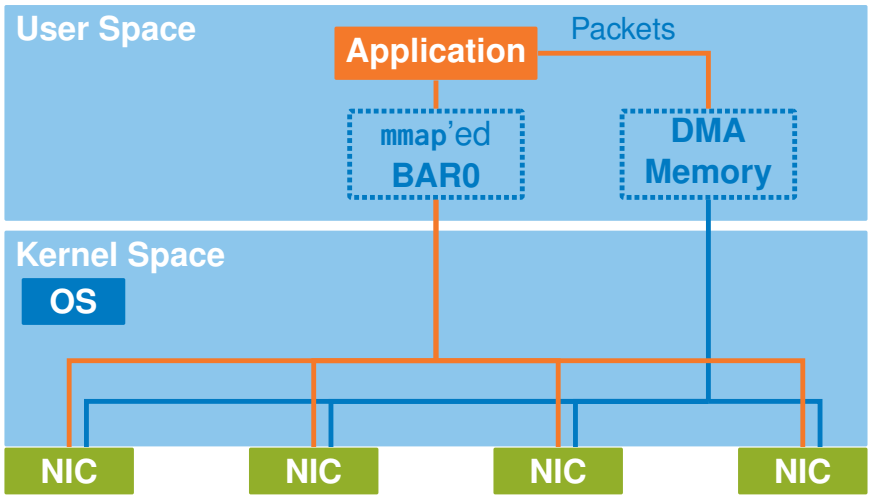


Figure 3.2: Architecture of full user space network drivers, e.g., DPDK, Snabb, or ixy.

language to those that can target eBPF bytecode (typically, a subset of C is used [27]). It is a default part of the Linux kernel nowadays and hence very well integrated. It is well-suited to implement firewalls that need to pass on traffic to the network stack [55]. More complex applications can be built on top of it with AF_XDP sockets, resulting in an architecture similar to netmap applications. Despite being part of the kernel, XDP does not yet work with all drivers as it requires a new memory model for all supported drivers. At the time of writing, XDP in kernel 5.15 (current long-term support kernel) supports fewer drivers than DPDK [81, 35].

DPDK [96], Snabb [99], and ixy implement the driver completely in user space. DPDK still uses a small kernel module with some drivers, but it does not contain driver logic and is only used during initialization. Snabb and ixy require no kernel code at all, see Figure 3.2. A main advantage of the full user space approach is that the application has full control over the driver leading to a far better integration of the application with the driver and the hardware. DPDK features the largest selection of offloading and filtering features of all investigated frameworks [36]. The downside is the poor integration with the kernel, DPDK's KNI (kernel network interface) needs to copy packets to pass them to the kernel unlike XDP or netmap which can just pass a pointer. Other advantages of DPDK are its support in the industry, mature code base, and large community. DPDK supports virtually all NICs commonly found in servers [35], far more than any other framework we investigated here.

ixy is a full user space driver as we want to explore writing drivers and not interfacing with existing drivers. Our architecture is based on ideas from both DPDK and Snabb.

The initialization and operation without loading a driver is inspired by Snabb, the API based on explicit memory management, batching, and driver abstraction is similar to DPDK.

3.3 Design

The language of choice for the explanation here and initial implementation is C as the lowest common denominator of systems programming languages. Our design goals are:

- Simplicity. A forwarding application including a driver should be less than 1,000 lines of C code.
- No dependencies. One self-contained project including the application and driver.
- Usability. Provide a simple-to-use interface for applications built on it.
- Speed. It should be reasonable fast without compromising simplicity, find the right trade-off.

It should be noted that the Snabb project [99] has similar design goals; ixy tries to be one order of magnitude simpler. For example, Snabb targets 10,000 lines of code [85], we target 1,000 lines of code and Snabb builds on Lua with LuaJIT instead of C limiting accessibility.

3.3.1 Architecture

ixy only features one abstraction level: it decouples the used driver from the user's application. Applications call into ixy to initialize a network device by its PCI address, ixy choses the appropriate driver automatically and returns a struct containing function pointers for driver-specific implementations. We currently expose packet reception, transmission, and device statistics to the application. Packet APIs are based on explicit allocation of buffers from specialized *memory pool* data structures.

Applications include the driver directly, ensuring a quick turn-around time when modifying the driver. This means that the driver logic is only a single function call away from the application logic, allowing the user to read the code from a top-down level without jumping between complex abstraction interfaces or even system calls.

3.3.2 NIC Selection

ixy is based on custom user space re-implementation of the Intel ixgbe driver cut down to their bare essentials. We tested our ixgbe driver on Intel X550, X540, and 82599ES NICs. All other frameworks except DPDK are also restricted to very few NIC models

(typically 3 or fewer families) and ixgbe is (except for OpenOnload only supporting their own NICs) always supported.

We chose ixgbe for ixy because Intel releases extensive datasheets and the ixgbe NICs are commonly found in commodity servers. These NICs are also interesting because they expose a relatively low-level interface to the drivers. Other NICs like the newer Intel XL710 series or Mellanox ConnectX-4/5 follow a more firmware-driven design: a lot of functionality is hidden behind a black-box firmware running on the NIC and the driver merely communicates via a message interface with the firmware which does the hard work. This approach has obvious advantages such as abstracting hardware details of different NICs allowing for a simpler more generic driver. However, our goal with ixy is understanding the full stack—a black-box firmware is counterproductive when the goal is to understand the inner workings.

We also implemented a driver for virtual VirtIO NICs tested against VirtualBox and qemu with and without vhost-user. VirtIO was selected as second driver to ensure that everyone can run the code without hardware dependencies. However, its performance proved to be too low to be interesting in the context of this dissertation, the main bottleneck was identified to be the crossing of the VM/hypervisor barrier.

3.3.3 User Space Drivers in Linux

There are two subsystems in Linux that enable user space drivers: `uio` and `vfio`, we support both.

`uio` exposes all necessary interfaces to write full user space drivers via memory mapping files in the `sysfs` pseudo filesystem. These file-based APIs give us full access to the device without needing to write any kernel code. ixy unloads any kernel driver for the given PCI device to prevent conflicts, i.e., there is no driver loaded for the NIC while ixy is running.

`vfio` offers more features: IOMMU and interrupts are only supported with `vfio`. However, these features come at the cost of additional complexity: It requires binding the PCIe device to the generic `vfio-pci` driver and it then exposes an API via `ioctl` syscalls on special files.

One needs to understand how a driver communicates with a device to understand how a driver can be written in user space. A driver can communicate via two ways with a PCIe device: The driver can initiate an access to the device's Base Address Registers (BARs) or the device can initiate a direct memory access (DMA) to access arbitrary main memory locations. BARs are used by the device to expose configuration and control registers to the drivers. These registers are available either via memory mapped

IO (MMIO) or via x86 IO ports depending on the device, the latter way of exposing them is deprecated in PCIe [130].

Accessing Device Registers

MMIO maps a memory area to device IO, i.e., reading from or writing to this memory area receives/sends data from/to the device. `uio` exposes all BARs in the `sysfs` pseudo filesystem, a privileged process can simply `mmap` them into its address space. `vfio` provides an `ioctl` that returns memory mapped to this area. Devices expose their configuration registers via this interface where normal reads and writes can be used to access registers. For example, ixgbe NICs expose all configuration, statistics, and debugging registers via the BAR0 address space. Our implementations of these mappings are in `pci_map_resource()` in `pci.c` and in `vfio_map_region()` in `libixy-vfio.c`.

A potential pitfall is that the exact size of the read and writes are important, e.g., accessing a single 32 bit register with two separate 16 bit reads will typically fail and trying to read multiple small registers with one read might not be supported. The exact semantics are up to the device, Intel's ixgbe NICs only expose 32 bit registers that support partial reads (except clear-on-read registers) but not partial writes.

DMA in User Space

DMA is initiated by the PCIe device and allows it to read/write arbitrary physical addresses. This is used to access packet data and to transfer the DMA descriptors (pointers to packet data) between driver and NIC. DMA needs to be explicitly enabled for a device via the PCI configuration space, our implementation is in `enable_dma()` in `pci.c` for `uio` and in `vfio_enable_dma()` in `libixy-vfio.c` for `vfio`. DMA memory allocation differs significantly between `uio` and `vfio`.

uio DMA memory allocation: Memory used for DMA transfer must stay resident in physical memory. `mlock(2)` [87] can be used to disable swapping. However, this only guarantees that the page stays backed by memory, it does not guarantee that the physical address of the allocated memory stays the same. The linux page migration mechanism can change the physical address of any page allocated by the user space at any time, e.g., to implement transparent huge pages and NUMA optimizations [97]. Linux does not implement page migration of explicitly allocated huge pages (2 MiB or 1 GiB pages on x86). ixy therefore uses huge pages which also simplify allocating physically contiguous chunks of memory. Huge pages allocated in user space are used by all investigated full user space drivers, but they are often passed off as a mere performance improvement [69, 153] despite being crucial for reliable allocation of DMA memory.

The user space driver hence also needs to be able to translate its virtual addresses to physical addresses, this is possible via the `procfs` file `/proc/self/pagemap`, the translation logic is implemented in `virt_to_phys()` in `memory.c`.

vfio DMA memory allocation: The previous DMA memory allocation scheme is specific to a quirk in Linux on x86 and not portable. `vfio` features a portable way to allocate memory that internally calls `dma_alloc_coherent()` in the kernel like an in-kernel driver would. This syscall abstracts all the messy details and is implemented in our driver in `vfio_map_dma()` in `libixy-vfio.c`. It requires an IOMMU and configures the necessary mapping to use virtual addresses for the device.

DMA and cache coherency: Both of our implementations require a CPU architecture with cache-coherent DMA access. Older CPUs might not support this and require explicit cache flushes to memory before DMA data can be read by the device. Modern CPUs do not have that problem. In fact, one of the main enabling technologies for high speed packet IO is that DMA accesses do not actually go to memory but to the CPU's cache on any recent CPU architecture.

Interrupts in User Space

`vfio` features full support for interrupts, `vfio_setup_interrupt()` in `libixy-vfio.c` enables a specific interrupt for vfio and associates it with an eventfd file descriptor. `enable_msix_interrupt()` in `ixgbe.c` configures interrupts for a queue on the device.

Interrupts are mapped to a file descriptor on which the usual syscalls like `epoll` are available to sleep until an interrupt occurs, see `vfio_epoll_wait()` in `libixy-vfio.c`.

3.3.4 Memory Management

ixy builds on an API with explicit memory allocation similar to DPDK which is a very different approach from netmap [148] that exposes a replica[1] of the NIC's ring buffer to the application. Memory allocation for packets was cited as one of the main reasons why netmap is faster than traditional in-kernel processing [148]. Hence, netmap lets the application handle memory allocation details. Many forwarding cases can then be implemented by simply swapping pointers in the rings. However, more complex scenarios where packets are not forwarded immediately to a NIC (e.g., because they are passed to a different core in a pipeline setting) do not map well to this API and require

[1] Not the actual ring buffers to prevent user-space applications from crashing the kernel with invalid pointers.

adding manual buffer management on top of this API. Further, a ring-based API is very cumbersome to use compared to one with memory allocation.

It is true that memory allocation for packets is a significant overhead in the Linux kernel, we have measured a per-packet overhead of 102 cycles when forwarding packets with Open vSwitch on Linux for allocating and freeing packet memory (measured with `perf`, more details in [40]). Note that forwarding 10 Gbit/s with minimum-sized packets on a single 2.4 GHz CPU core leaves a budget of only 161 cycles/packet in total, see Section 2.2. That means more than half of the available time is spent on memory management overhead in the Linux kernel.

This overhead is almost completely due to (re-)initialization of the kernel `sk_buff` struct: a large data structure with a lot of metadata fields targeted at a general-purpose network stack. Linux mixes high-level protocol logic with low-level driver and memory management logic: Every protocol supported by the kernel stores data in this struct, leading to a more complex re-initialization process (more/bigger writes) on a larger struct (poorer memory locality). Memory allocation in ixy (with minimum metadata required for the driver) only adds an overhead of 30 cycles/packet. The idea is that you only pay for the features you need, the buffer is extensible and you can add your own additional metadata if necessary.

ixy's API is the same as DPDK's API when it comes to sending and receiving packets and managing memory. It can best be explained by reading the example applications `ixy-fwd.c` and `ixy-pktgen.c`. The transmit-only example `ixy-pktgen.c` creates a *memory pool*, a fixed-size collection of fixed-size packet buffers and pre-fills them with packet data. It then allocates a batch of packets from this pool, adds a sequence number to the packet, and passes them to the transmit function. The transmit function is asynchronous: it enqueues pointers to these packets, the NIC fetches and sends them later. Previously sent packets are freed asynchronously in the transmit function by checking the queue for sent packets and returning them to the pool. This means that a packet buffer cannot be re-used immediately, the `ixy-pktgen` example looks therefore quite different from a packet generator built on a classic socket API.

The forward example `ixy-fwd.c` can avoid explicit handling of memory pools in the application: the driver allocates a memory pool for each receive ring and automatically allocates packets. Allocation is done by the packet reception function, freeing is either handled in the transmit function as before or by dropping the packet explicitly if the output link is full. Exposing the rings directly similar to netmap could significantly speed up this simple example application at the cost of usability.

3.3.5 Security Considerations

User space drivers effectively run with root privileges even if they drop privileges after initializing devices: they can use the device's DMA capabilities to access arbitrary memory locations, negating some of the security advantages of running in user space. This can be mitigated by using the IO memory management unit (IOMMU) to isolate the address space accessible to a device at the cost of an additional (hardware-accelerated) lookup in a page table for each memory access by the device.

IOMMUs are available on CPUs offering hardware virtualization features as they were designed to pass PCIe devices (or parts of them via SR-IOV) directly into VMs in a secure manner. Linux abstracts different IOMMU implementations via the `vfio` framework which is specifically designed for "safe non-privileged userspace drivers" [98] beside virtual machines. Our `vfio` backend allows running the driver and application as an unprivileged user. Of the investigated other frameworks only netmap supports this. DPDK also offers a `vfio` backend and has historically supported running with unprivileged users, but recent versions no longer support this with most drivers. Snabb's `vfio` backend was removed because of the high maintenance burden and low usage.

3.4 ixgbe Implementation

All page numbers and section numbers for the Intel datasheet refer to revision 3.3 of the 82599ES datasheet [73].

ixgbe devices expose all configuration, statistics, and debugging registers via the BAR0 MMIO region. The datasheet lists all registers as offsets in this configuration space in Section 9 [73]. We use `ixgbe_type.h` from Intel's driver as a machine-readable version of the datasheet, it contains defines for all register names and offsets for bit fields. This is technically a violation of both our goals about dependencies and lines of code, but we only effectively use less than 100 lines that are just defines and simple structs. There is nothing to be gained from manually copying offsets and names from the datasheet or this file.

3.4.1 NIC Ring API

NICs expose multiple circular buffers called queues or rings to transfer packets. The simplest setup uses only one receive and one transmit queue. Multiple transmit queues are merged on the NIC, incoming traffic is split according to filters or a hashing algorithm if multiple receive queues are configured. Both receive and transmit rings work in a similar way: the driver programs a physical base address and the size of the ring. It then fills the memory area with *DMA descriptors*, i.e., pointers to physical addresses

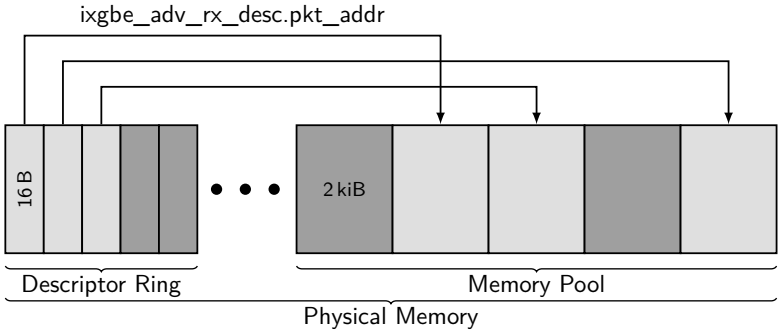


Figure 3.3: DMA descriptors pointing into a memory pool, note that the packets in the memory are unordered as they can be free'd at different times.

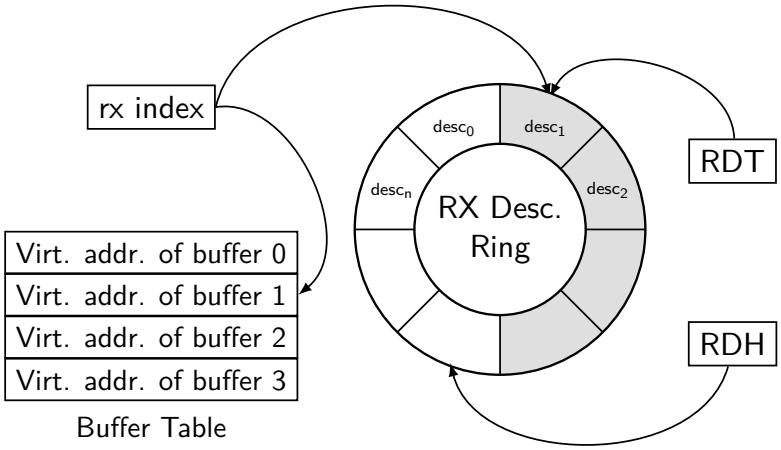


Figure 3.4: Overview of a receive queue. The ring uses physical addresses and is shared with the NIC.

where the packet data is stored with some metadata. Sending and receiving packets is done by passing ownership of the DMA descriptors between driver and hardware via a head and a tail pointer. The driver controls the tail, the hardware the head. Both pointers are stored in device registers accessible via MMIO.

The initialization code is in `ixgbe.c` starting from line 114 for receive queues and from line 173 for transmit queues. Further details are in the datasheet in Section 7.1.9 and in the datasheet sections mentioned in the code.

Receiving Packets

The driver fills up the ring buffer with physical pointers to packet buffers in `start_rx_queue()` on startup. Each time a packet is received, the corresponding buffer is returned to the application and we allocate a new packet buffer and store its physical address in the DMA descriptor and reset the ready flag. We also need a way to translate

the physical addresses in the DMA descriptor found in the ring back to its virtual counterpart on packet reception. This is done by keeping a second copy of the ring populated with virtual instead of physical addresses, this is then used as a lookup table for the translation.

Figure 3.3 illustrates the memory layout: the DMA descriptors in the ring to the left contain physical pointers to packet buffers stored in a separate location in a memory pool. The packet buffers in the memory pool contain their physical address in a metadata field. Figure 3.4 shows the RDH (head) and RDT (tail) registers controlling the ring buffer on the right side, and the local copy containing the virtual addresses to translate the physical addresses in the descriptors in the ring back for the application. `ixgbe_rx_batch()` in `ixgbe.c` implements the receive logic as described by Sections 1.8.2 and 7.1 of the datasheet. It operates on batches of packets to increase performance. A naïve way to check if packets have been received is reading the head register from the NIC incurring a PCIe round trip. The hardware also sets a flag in the descriptor via DMA which is far cheaper to read as the DMA write is handled by the last-level cache on modern CPUs. This is effectively the difference between an LLC cache miss and hit for every received packet.

Transmitting Packets

Transmitting packets follows the same concept and API as receiving them, but the function is more complicated because the interface between NIC and driver is asynchronous. Placing a packet into the ring does not immediately transfer it and blocking to wait for the transfer is infeasible. Hence, the `ixgbe_tx_batch()` function in `ixgbe.c` consists of two parts: freeing packets from previous calls that were sent out by the NIC followed by placing the current packets into the ring. The first part is often called cleaning and works similar to receiving packets: the driver checks a flag that is set by the hardware after the packet associated with the descriptor is sent out. Sent packet buffers can then be free'd, making space in the ring. Afterwards, the pointers of the packets to be sent are stored in the DMA descriptors and the tail pointer is updated accordingly.

Checking for transmitted packets can be a bottleneck due to cache thrashing as both the device and driver access the same memory locations [73]. The 82599 hardware implements two methods to combat this: marking transmitted packets in memory occurs either automatically in configurable batches on device side, this can also avoid unnecessary PCIe transfers. We tried different configurations (code in `init_tx()`) and found that the defaults from Intel's driver work best. The NIC can also write its current position in the transmit ring back to memory periodically (called head pointer write back) as explained in Section 7.2.3.5.2 of the datasheet. However, no other driver im-

plements this feature despite the datasheet referring to the normal marking mechanism as "legacy". We implemented support for head pointer write back on a branch [102] but found no measurable performance improvements or effects on cache contention.

Batching

Each successful transmit or receive operation involves an update to the NIC's tail pointer register (RDT or TDT for receive/transmit), a slow operation. This is one of the reasons why batching is so important for performance. Both the receive and transmit function are batched in ixy, updating the register only once per batch.

Offloading Features

ixy currently only enables CRC checksum offloading. Unfortunately, packet IO frameworks (e.g., netmap) are often restricted to this bare minimum of offloading features. DPDK is the exception here as it supports almost all offloading features offered by the hardware. However, its receive and transmit functions pay the price for these features in the form of complexity.

We will try to find a balance and showcase selected simple offloading features in ixy in the future. These offloading features can be implemented in the receive and transmit functions, see comments in the code. This is simple for some features like VLAN tag offloading and more involved for more complex features requiring an additional descriptor containing metadata information.

3.5 Performance Evaluation

Recall the key performance numbers from Section 2.2: We want to handle 14.88 million packets per second (Mpps) on a single core, leaving us with a time budget of 67.2 ns per packet. ixy is much faster faster than that, so we increase the load and use two 10 Gbit/s ports (29.76 Mpps with minimum-sized packets) at the same time on the same CPU core. We compare ixy's `ixy-fwd` and compare it to a custom DPDK-based forwarder implementing the same features.

Both forwarders move packets bidirectionally between two ports and modify a byte in the packet to simulate a somewhat realistic workload. Not doing so would be highly unrealistic: Loading data from the L3 cache (where the NIC puts the packet via DMA) into the actual core of the CPU takes 53 clock cycles [109], a significant fraction of the available time budget. We use an Intel Xeon E5-2620 v3 2.4 GHz CPU here with several different Intel NICs based on the ixgbe family (Server 1 from Section 2.4). Section 2.1 gives an overview over this architecture.

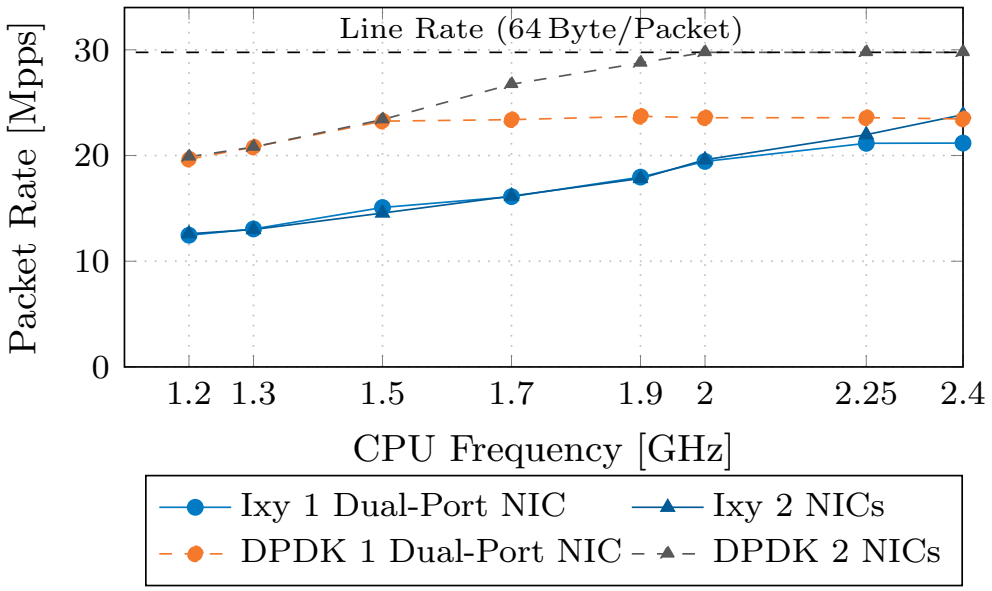


Figure 3.5: Bidirectional single-core forwarding performance with varying CPU speed, batch size 32.

3.5.1 Methodology

We use the MoonGen packet generator developed for this dissertation (Chapter 5), loading the system with the full rate of 29.76 Mpps for all experiments unless mentioned otherwise in the description of a test setup. All measurements of throughput are taken by loading the system for at least 10 seconds and taking an average throughput observed by the packet generator every second. The packet rate varies by less than 0.1% both within a single run and between different runs, so we omit error bars in the diagrams. Latency measurements are based on MoonGen's hardware timestamping (see Section 5.6).

3.5.2 Throughput

To quantify the baseline performance and identify bottlenecks, we run the forwarding example while increasing the CPU's clock frequency from 1.2 GHz to 2.4 GHz. Figure 3.5 compares the throughput of our forwarder on ixy and on DPDK when forwarding across the two ports of a dual-port NIC and when using two separate NICs.

The better performance of both ixy and DPDK when using two separate NICs over one dual-port NIC indicates a hardware limit. This bottleneck likely happens at the PCIe level, but we were never able to confirm this with the available performance counters. We run this test on Intel X520 (82599-based) and Intel X540 NICs with identical results. ixy requires 96 CPU cycles to forward a packet, DPDK only 61.

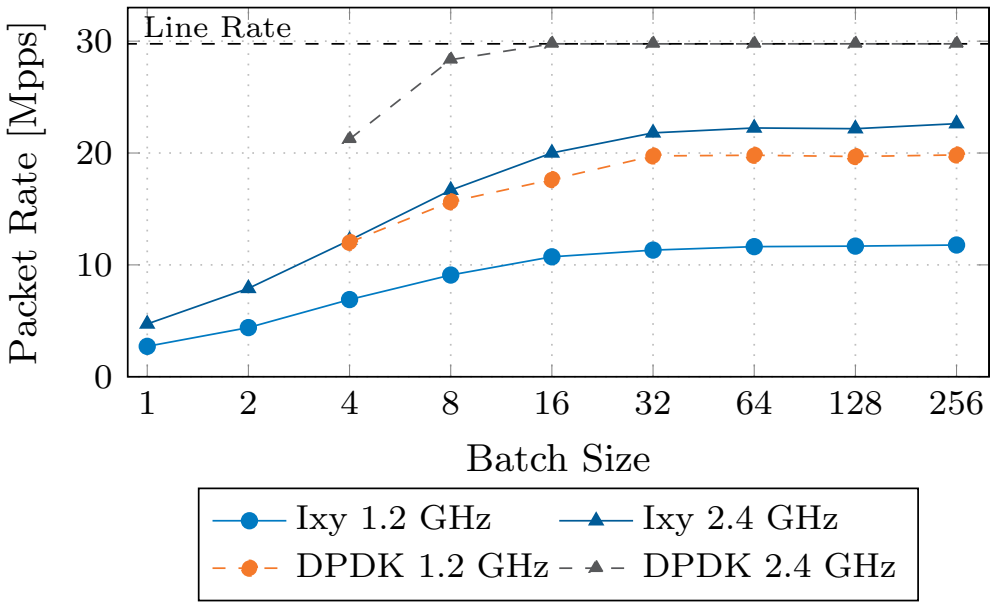


Figure 3.6: Bidirectional single-core forwarding performance with varying batch size.

The high performance of DPDK can be attributed to its more complex transmit path. As noted in the beginning of this chapter, DPDK features 5,400 lines of code whereas ixy only needs 127 lines. This more complex code features multiple different implementations of the transmit logic, each specialized for particular requirements. In this experiment DPDK picked a vectorized code path specialized to the SIMD instructions available on this CPU. It could only do so because we did not enable offloading features at device configuration time (meaning the feature set is similar to ixy). DPDK also requires us to use a batch size of at least 4 when used in this mode.

Disabling this transmit path in the DPDK configuration, or using an older version of DPDK, increases the CPU cycles per packet to 91 cycles packet, still slightly faster than ixy despite doing more (checking for more offloading flags). Overall, we consider ixy fast enough for our purposes. For comparison, performance evaluations of older (2015) versions of DPDK, PF_RING, and netmap and required ≈100 cycles/packet for DPDK and PF_RING and ≈120 cycles/packet for netmap [51].

3.5.3 Batching

Batching is one of the main drivers for performance [52, 89, 14]. DPDK even requires a minimum batch size of 4 when using the SIMD transmit path. Receiving or sending a packet involves an access to the queue index registers, invoking a costly PCIe round-trip. Figure 3.6 shows how the performance increases as the batch size is increased in the bidirectional forwarding scenario with two NICs. Increasing batch sizes have

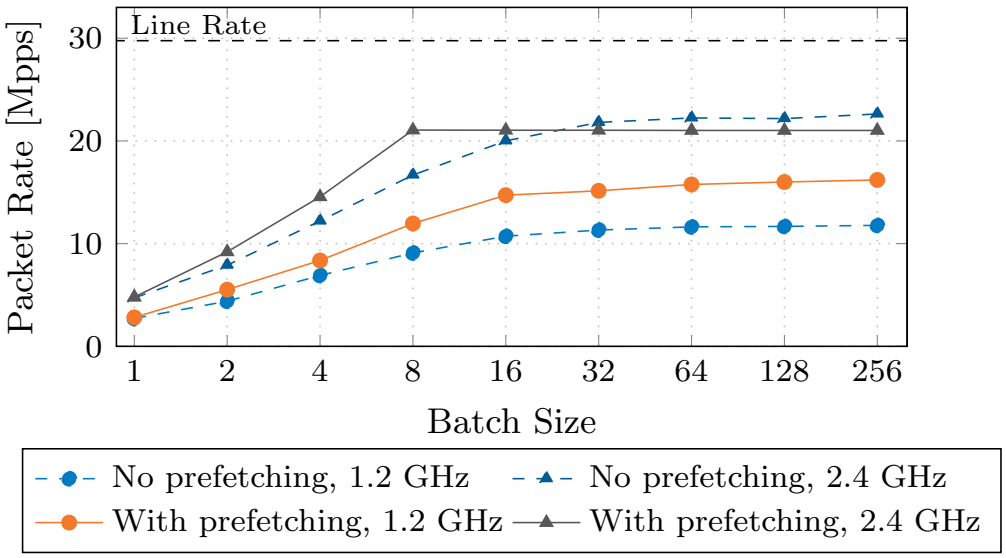


Figure 3.7: Effect of prefetching on bidirectional single-core forwarding performance with varying batch size.

diminishing returns: this is especially visible when the CPU is only clocked at 1.2 GHz. Reading the performance counters for all caches shows that the number of L1 cache misses per packet increases as the performance gains drop off. Too large batches thrash the L1 cache, possibly evicting lookup data structures in a real application. Therefore, batch sizes should not be chosen too large. We conclude that a batch size of between 32 and 64 should be selected. Latency is also impacted by the batch size, but the effect is negligible compared to other buffers (e.g., NIC ring size of 512).

3.5.4 Memory Prefetching

DMA writes by the NIC are stored in the CPU's L3 cache by the memory controller (cf. Section 2.1), so traditional memory prefetching from main memory to the caches cannot be applied. However, memory prefetching is also effective when applied to different layers of a cache hierarchy. We implemented prefetching in the receive loop of the driver to evaluate its effect, the code is found on the branch `prefetching`. The driver accesses all DMA descriptors within a batch, each DMA descriptor points to a memory buffer containing both metadata and data. Applications on top of the driver will then only access the memory buffer itself; this indirection between DMA descriptor and packet buffer is an opportunity for prefetching.

Figure 3.7 shows the effects of explicitly prefetching packet data in the receive loop of a driver on the forwarding application running the same benchmark as the previous section. Prefetching at a batch size of 1 has no measurable effect because the access on

Intr./polling	Load	Median	99th perc.	99.99th perc.	Max
Polling	0.1 Mpps	3.8 µs	4.7 µs	5.8 µs	15.8 µs
Intr., no throttling	0.1 Mpps	7.7 µs	8.4 µs	11.0 µs	76.6 µs
Intr., ITR 10 µs	0.1 Mpps	11.3 µs	11.9 µs	15.3 µs	78.4 µs
Intr., ITR 200 µs	0.1 Mpps	107.4 µs	208.1 µs	240.0 µs	360.0 µs
Polling	0.4 Mpps	3.8 µs	4.6 µs	5.8 µs	16.4 µs
Intr., no throttling	0.4 Mpps	7.3 µs	8.2 µs	10.9 µs	53.9 µs
Intr., ITR 10 µs	0.4 Mpps	9.0 µs	14.0 µs	25.8 µs	86.1 µs
Intr., ITR 200 µs	0.4 Mpps	105.9 µs	204.6 µs	236.7 µs	316.2 µs
Polling	0.8 Mpps	3.8 µs	4.4 µs	5.6 µs	16.8 µs
Intr., no throttling	0.8 Mpps	5.6 µs	8.2 µs	10.8 µs	81.1 µs
Intr., ITR 10 µs	0.8 Mpps	9.2 µs	14.1 µs	31.0 µs	70.2 µs
Intr., ITR 200 µs	0.8 Mpps	102.8 µs	198.8 µs	226.1 µs	346.8 µs

Table 3.1: Forwarding latency by interrupt/poll mode.

the data by the application immediately follows the access on the DMA descriptor by the driver. Larger batch sizes lead to a larger temporal difference between these two accesses giving the CPU the necessary time to complete the prefetch in the background. We observe a performance increase of up to 31% for the somewhat unrealistic case of the 1.2 GHz CPU. There is one scenario in which explicit prefetching harms performance: when hitting the aforementioned hardware bottleneck at ≈22 Mpps. Inserting the prefetch instruction drops performance by up to 7% in this case, indicating that the prefetch is being executed in the same hardware resource that causes the bottleneck.

We conclude that explicit prefetching is not crucial for performance in realistic scenarios (batch size 32 to 64, CPU at full clock speed) on the Haswell EP CPU platform evaluated here. The CPU's heuristics about implicitly prefetching are sufficient since the packets are stored and handled sequentially in an easily predictable pattern.

3.5.5 Interrupts

Interrupts are a common mechanism to reduce power consumption at low loads. However, interrupts are expensive: they require multiple context switches. This makes them unsuitable for high packet rates. NICs commonly feature interrupt throttling (ITR, configured in µs between interrupts on the NIC used here) to prevent overloading the system. Operating systems often disable interrupts periodically and switch to polling during periods of high loads (e.g., Linux NAPI, for more details please refer to [16]). Our forwarder loses packets in interrupt mode at rates of above around 1.5 Mpps even with aggressive throttling configured on the NIC. All other tests except this one are therefore conducted in pure polling mode.

A common misconception is that interrupts reduce latency, but they actually increase latency. The reason is that an interrupt first needs to wake the CPU from low-power states (power states down to C6 are enabled on the test system), trigger a context switch into interrupt context, trigger another switch to the driver and then poll the packets from the NIC[1]. Note that if the interrupt rate is sufficiently high such that the system is not an idle state when the interrupt arrives, then this penalty does not apply. Using interrupts in this case is just unnecessary overhead in this case which is why Linux NAPI does switch to polling under higher load. Yet, handling the packet with the interrupt is still slower than permanently polling: the interrupt is additional work, it still needs to poll after receiving the interrupt.

Table 3.1 shows latencies at low rates (where interrupts are effective) with and without interrupt throttling (ITR) and polling. Especially tail latencies are affected by using interrupts instead of polling. All timestamps were acquired with a fiber-optic splitter and MoonGen taking a hardware timestamp of every single packet. Each measurement was run for 10 seconds, so the data is based on 1,000,000 to 8,000,000 timestamps as the load increases from 0.1 Mpps to 0.8 Mpps.

These results show that interrupts with a low throttle rate are feasible at low packet rates. Interrupts are poorly supported in other user space drivers: Snabb offers no interrupts, DPDK has limited support for interrupts (only some drivers) without built-in automatic switching between different modes. Frameworks relying on a kernel driver can use the default driver's interrupt features, especially netmap offers good support for power-saving via interrupts.

3.5.6 Profiling

We run `perf` on `ixy-fwd` running under full bidirectional load at 1.2 GHz with two different NICs using the default batch size of 32 to ensure that the CPU is the only bottleneck. The performance scales linearly with CPU frequency with these settings in this frequency range, so the CPU is the limiting factor. `perf` allows profiling with the minimum possible effect on the performance: it periodically samples the current execution state of a given program and derives utilization from this. We ran perf for 5 minutes with a sampling frequency of 997 Hz, a prime number to avoid overlap with other period processes. Throughput drops by only ≈5% while `perf` is running.

[1] The same is true for Linux kernel drivers, the actual packet reception is not done in the hardware interrupt context but in a software interrupt

App/Function	RX	TX	Forwarding	Memory Mgmt.
ixy-fwd	44.8	14.7	12.3	30.4
ixy-fwd-inline	57.0	28.3	12.5	?*
DPDK v17.11 l2fwd	35.4	20.4	†6.1	?*
DPDK v1.6 l2fwd‡	41.7	53.7	†6.0	?*

*Memory operations inlined, separate profiling not possible.
†DPDK's driver explicitly prefetches packet data on RX, so this is faster despite performing the same action of changing one byte.
‡Old version 1.6 (2014) of DPDK, far fewer SIMD optimizations, measured on a different system/kernel due to compatibility.

Table 3.2: Processing time in cycles per packet.

However, we have no way of guaranteeing that this overhead is distributed equally across all functions.

Table 3.2 shows where CPU time is spent on average per forwarded packet and compares it to DPDK. Receiving is slower because the receive logic performs the initial fetch, the following functions operate on the L1 cache. ixy's receive function still leaves room for improvements, it is less optimized than the transmit function. There are several places in the receive function where DPDK avoids memory accesses by batching compared to ixy. However, these optimizations were not applied for simplicity in ixy: DPDK's receive function is quite complex and full of SIMD intrinsics leading to poor readability. We also compare an old version of DPDK in the table that did not yet contain as many optimizations; ixy outperforms the old DPDK version on the under-clocked CPU.

Overhead for memory management is significant (but still low compared to the 101 cycles/packet in the Linux kernel from Section 3.3.4). 59% of the time is spent in non-batched memory operations and none of the calls are inlined. Inlining these functions increases throughput by 6.5% but takes away our ability to account time spent in them. Overall, the overhead of memory management is larger than we initially expected, but we still think explicit memory management for the sake of a usable API is a good trade-off. This is especially true for ixy aiming at simplicity, but also for other frameworks targeting complex applications. Simple forwarding can easily be done on an exposed ring interface, but anything more complex that does not sent out packets immediately (e.g., because they are processed further on a different core) requires memory management in the user's application with a similar performance impact.

3.5.7 Queue Sizes

Our driver supports descriptor ring sizes in power-of-two increments between 64 and 4096, the hardware supports more sizes but the restriction to powers of two simplifies wrap-around handling. Linux defaults to a ring size of 256 for this NIC, DPDK's

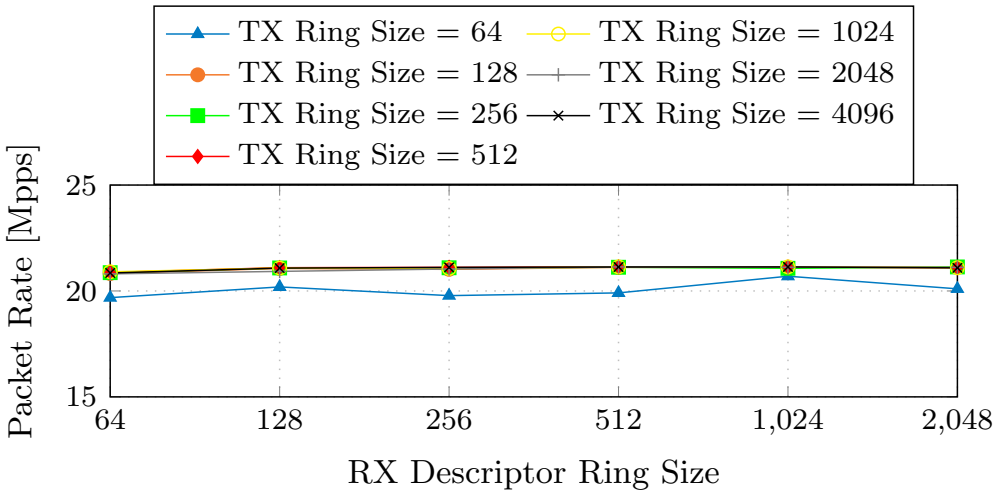


Figure 3.8: Throughput (64 Byte packets) with varying descriptor ring sizes at 2.4 GHz.

example applications configure different sizes; the `l2fwd` forwarder sets 128/512 RX/TX descriptors. Larger ring sizes such as 8192 are sometimes recommended to increase performance [12] (source refers to the size as kB when it is actually number of packets). Figure 3.8 shows the throughput of ixy with various ring size combinations. There is no measurable impact on the maximum throughput for ring sizes larger than 64. Scenarios where a larger ring size can still be beneficial might exist: for example, an application producing a large burst of packets significantly faster than the NIC can handle for a very short time.

The second performance factor that is impacted by ring sizes is the overall latency caused by unnecessary buffering. Table 3.3 shows the latency of the ixy forwarder with different ring sizes. Latency was measured by sampling 1,000 packets per second with MoonGen over a period of at least 180 seconds.

The results show a linear dependency between ring size and latency when the system is overloaded, but the effect under lower loads are negligible. Full or near full buffers are

Ring Sizes	Load	Median	99th perc.	99.9th perc.
64	15 Mpps	5.2 µs	6.4 µs	7.2 µs
512	15 Mpps	5.2 µs	6.5 µs	7.5 µs
4096	15 Mpps	5.4 µs	6.8 µs	8.7 µs
64	*29 Mpps	8.3 µs	9.1 µs	10.6 µs
512	*29 Mpps	50.9 µs	52.3 µs	54.3 µs
4096	*29 Mpps	424.7 µs	433.0 µs	442.1 µs

*Device under test overloaded, packets were lost

Table 3.3: Forwarding latency by ring size and load.

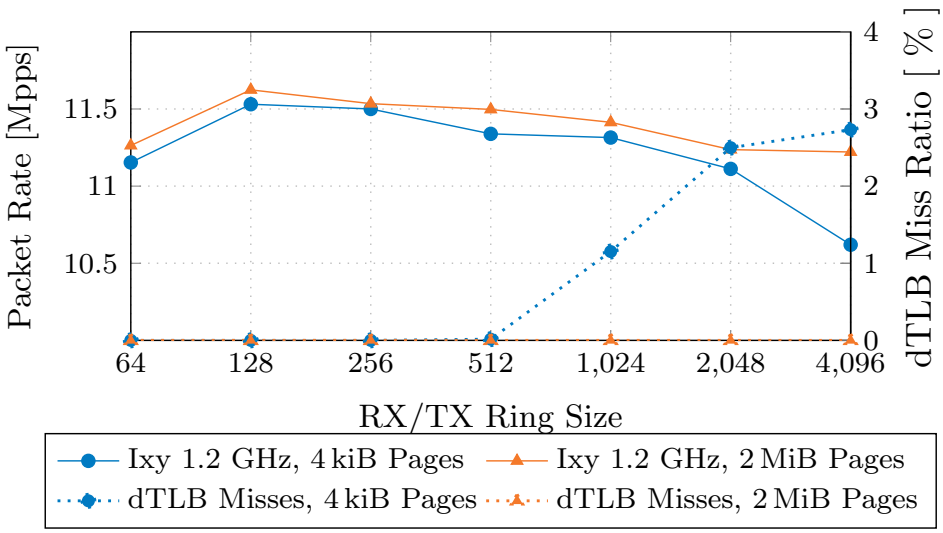


Figure 3.9: Single-core forwarding performance with and without huge pages and their effect on the TLB.

no exception on systems forwarding Internet traffic due to protocols like TCP that try to fill up buffers completely [54]. We conclude that tuning tips like setting ring sizes to 8192 [12] are detrimental for latency and likely do not help with throughput. ixy uses a default ring size of 512 at the moment as a trade-off between providing some buffer and avoiding high worst-case latencies.

3.5.8 Page Sizes without IOMMU

It is not possible to allocate DMA memory on small pages from user space in Linux in a reliable manner without using the IOMMU as described in Section 3.3.3. Despite this, we have implemented an allocator that performs a brute-force search for physically contiguous normal-sized pages from user space. We run this code on a system without NUMA and with transparent huge pages and page-merging disabled to avoid unexpected page migrations. The code for these benchmarks is hidden on a branch [101] due to its unsafe nature on some systems (we did lose a file system to rogue DMA writes on a misconfigured server). Benchmarks varying the page size are interesting despite these problems: kernel drivers and user space packet IO frameworks using kernel drivers only support normal-sized pages. Existing performance claims about huge pages in drivers are vague and unsubstantiated [69, 153].

Figure 3.9 shows that the impact on performance of huge pages in the driver is small. The performance difference is 5.5% with the maximum ring size, more realistic ring sizes only differ by 1-3%. This is not entirely unexpected: the largest queue size of 4096

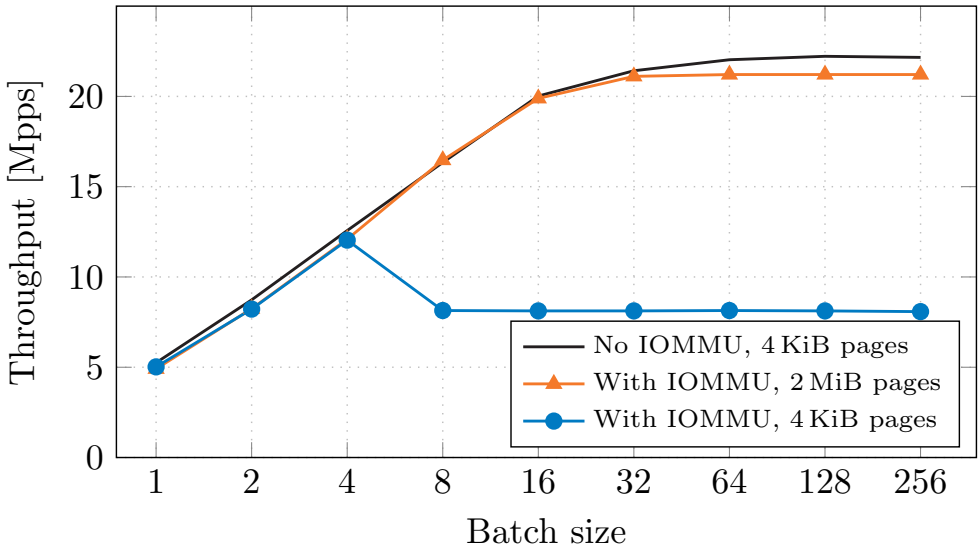


Figure 3.10: IOMMU impact on single-core forwarding at 2.4 GHz.

entries is only 16 kiB large, storing pointers to up to 16 MiB packet buffers. Huge pages are designed for, and usually used with, large data structures, e.g., big lookup tables for forwarding. The effect measured here is likely larger when a real forwarding application puts additional pressure on the TLB (4096 entries on the CPU used here) due to its other internal data structures. One should still use huge pages for other data structures in a packet processing application, but a driver not supporting them is not as bad as one might expect when reading claims about their importance from authors of drivers supporting them.

3.5.9 Page Sizes and IOMMU Overhead

Memory access overhead changes if the device has to go through the IOMMU for every access. Documentation for Intel's IOMMU is sparse: The TLB size is not documented and there are no dedicated performance counters. Neugebauer et al. experimentally determined a TLB size of 64 entries with their pcie-bench framework [112] (vs. 4096 entries in the normal TLB). Our observations are also consistent with a TLB size of only 64 entries. They note a performance impact for small DMA transactions with large window sizes: 64 byte read throughput drops by up to 70%, write throughput by up to 55%. 256 byte reads are 30% slower, only 512 byte and larger transactions are unaffected [112]. Their results are consistent across four different Intel microarchitectures including the CPU we are using here. They explicitly disable huge pages for their IOMMU benchmark.

Our benchmark triggers a similar scenario when not used with huge pages: We ask the NIC to transfer a large number of small packets via DMA. Note that packets in a batch are not necessarily contiguous in memory: Buffers are not necessarily allocated sequentially and each DMA buffer is 2 KiB large by default, of which only the first n bytes will be transferred. This means only two packets share a 4 kiB page, even if the packets are small. 2 KiB is a common default in other drivers as it allows handling normal sized frames without chaining several buffers (the NIC only supports DMA buffers that are a multiple of 1 kiB). The NIC therefore has to perform several small DMA transactions, i.e., the scenario is equivalent to a large transfer window in pcie-bench.

Figure 3.10 shows that the IOMMU does not affect the performance if used with 2 MiB pages. However, the default 4 KiB pages (that are safe and easy to use with `vfio` and the IOMMU) are affected by the small TLB in the IOMMU. The impact of the IOMMU on our real application is slightly smaller than in the synthetic pcie-bench tests: The IOMMU decreases throughput by 62% for the commonly used batch size of 32 with small packets when not using huge pages. Running the test with 128 byte packets decreases throughput by 33% and 256 byte packets are not affected at all.

However, enabling huge pages completely mitigates the impact of the small TLB in the IOMMU. Note that huge pages for the IOMMU are only supported since the Intel Ivy Bridge CPU generation.

3.5.10 NUMA Considerations

Non-uniform memory access (NUMA) architectures found on multi-CPU servers present additional challenges. Modern systems integrate cache, memory controller, and PCIe root complex in the CPU itself instead of using a separate IO hub. This means that a PCIe device is attached to only one CPU in a multi-CPU system, access from or to other CPUs needs to pass over the CPU interconnect (QuickPath Interconnect on our system). At the same time, the tight integration of these components allows the PCIe controller to transparently write DMA data into the cache instead of main memory. This works even when direct cache access (DCA) is not used (DCA is only supported by the kernel driver, none of the full user space drivers implement it). Intel Data Direct I/O (DDIO) is another optimization to prevent memory accesses by DMA [75]. However, we found by reading performance counters that even CPUs not supporting DDIO do not perform memory accesses in a typical packet forwarding scenario. DDIO is poorly documented and exposes no performance counters, its exact effect on modern systems is unclear. All recent (since 2012) CPUs supporting multi-CPU systems also support DDIO. Our NUMA benchmarks where obtained on a different system, Server 2 from

Ingress*	Egress*	CPU†	Memory‡	Throughput
Node 0	Node 0	Node 0	Node 0	10.8 Mpps
Node 0	Node 0	Node 0	Node 1	10.8 Mpps
Node 0	Node 0	Node 1	Node 0	7.6 Mpps
Node 0	Node 0	Node 1	Node 1	6.6 Mpps
Node 0	Node 1	Node 0	Node 0	7.9 Mpps
Node 0	Node 1	Node 0	Node 1	10.0 Mpps
Node 0	Node 1	Node 1	Node 0	8.6 Mpps
Node 0	Node 1	Node 1	Node 1	8.1 Mpps

*NUMA node connected to the NIC
†Thread pinned to this NUMA node
‡Memory pinned to this NUMA node

Table 3.4: Unidirectional forwarding on a NUMA system, both CPUs at 1.2 GHz.

Section 2.4, (2x Intel Xeon E5-2630 v4) than the previous results because we want to avoid potential problems with NUMA for the other setups.

Our test system has one dual-port NIC attached to NUMA node 0 and a second one to NUMA node 1. Both the forwarding process and the memory used for the DMA descriptors and packet buffers can be explicitly pinned to a NUMA node. This gives us 8 possible scenarios for unidirectional packet forwarding by varying the packet path and pinning. Table 3.4 shows the throughput at 1.2 GHz. Forwarding from and to a NIC at the same node shows one unexpected result: pinning memory, but not the process itself, to the wrong NUMA node does not reduce performance. The explanation for this is that the DMA transfer is still handled by the correct NUMA node to which the NIC is attached, the CPU then caches this data while informing the other node. However, the CPU at the other node never accesses this data and there is hence no performance penalty. Forwarding between two different nodes is fastest when the memory is pinned to the egress nodes and CPU to the ingress node and slowest when both are pinned to the ingress node. Real forwarding applications often cannot know the destination of packets at the time they are received, the best guess is therefore to pin the thread to the node local to the ingress NIC and distribute packet buffer across the nodes. Latency was also impacted by poor NUMA mapping, we measured an additional 1.7 μs when unnecessarily crossing the NUMA boundary.

3.6 Conclusions

The insights gained in this chapter address research question Q1 from Section 1.2: What makes software-based packet processing fast? We now have an understanding about the foundations and basic principles of fast packet processing in software.

Having a completely custom driver allows us to look at individual effects in isolation. Our key findings are that it is feasible to receive, modify, and forward a packet in only 96 clock cycles. It is crucial to have a batch size of at least 32 packets to achieve this, see Sections 3.5.2 and 3.5.3. Further, we find that the NIC's ring buffers should not be configured to more than 512 entries in order to keep the latency at below 100 μs even under overload conditions (Section 3.5.7). Performance is not impacted by using the IOMMU (to run without root privileges) if huge pages (2 MiB) are used, we measure a decrease of up to 62% in throughput with the default 4 KiB page size (see Section 3.5.9). We can now apply these learnings when going up in the stack in the following chapters.

The main takeaway from this chapter is the architecture of ixy (Section 3.3). ixy serves as a base for our high-level language driver implementations following in the next chapter. All following drivers are built on the exact same ideas to allow for a fair comparison. The experiments conducted here also serve as baseline for a comparison of drivers written in different languages for a quantitative comparison of different language features.

3.7 Author's Contributions

Sections 3.1 to 3.5 are based on the following publication [44]:

> Paul Emmerich, Maximilian Pudelko, Simon Bauer, Stefan Huber, Thomas Zwickl, and Georg Carle. "User Space Network Drivers". In: *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2019)*. IEEE. Cambridge, UK, Sept. 2019

The author of this dissertation came up with the design for the ixy framework and implemented its core together with the ixgbe (82599) driver. This includes hugepage-based memory management logic that works without using vfio, the batch-based API, and the ixgbe poll-mode driver itself. The author contributed to the IOMMU/vfio design.

All experiments and benchmarks presented here were conceived and fully specified by the author. The author also made contributions to the executions of the experiments and test runs. All white-box profiling experiments were done by the author. Evaluation and analysis of all experiment results was done by the author.

The following significant changes vs. the paper were made for this dissertation:

- Memory prefetching, an entirely new feature, was implemented for this dissertation.

- The evaluation of the effects of memory prefetching is new.
- References to our VirtIO implementation, which is not related to the main points of our thesis, have been removed.

Chapter 4

High-Level Languages for Network Drivers

Our driver presented in the last chapter is written in C and is a stepping stone in our journey towards understanding the use of high-level languages in the context of network applications. A traditional and simple implementation in C serves as a good baseline for comparisons with other languages as well as a universally readable reference implementation.

Understanding if and how high-level languages can be used for software-based packet processing systems (research question Q2) is the next step in our journey. We take an experimental approach in this dissertation: We re-implement the same driver in a variety of different languages and compare them.

Our goal for this chapter is to understand trade-offs: which language features are helpful for network systems? What is their cost? For example, a language with garbage collection avoids bugs related to memory safety (a common problem in C), but the garbage collector can stop our program at unpredictable times thus causing unpredictable latency. We observe tail latencies of several hundred microseconds in some garbage collected languages such as OCaml, Haskell and JavaScript (Section 4.7). Quantifying this and other effects allows us to make an informed choice about the language selection. We find that Rust is the ideal choice for a high-level language in drivers, we are able to write 87% of the driver code in memory-safe code (Section 4.4.3) while achieving the same latency and only sacrificing 2% throughput (Section 4.6.2).

The remainder of this chapter is based on our publication about high-level languages for drivers which is joint work by Paul Emmerich, Simon Ellmann, Fabian Bonk, Alex

Egger, Esaú García Sánchez-Torija, Thomas Günzel, Sebastian Di Luzio, Alexandru Obada, Maximilian Stadlmeier, Sebastian Voit, and Georg Carle [43]. A full account of the author's contributions is given in Section 4.9.

4.1 Introduction

C has been the go-to language for writing kernels since its inception. Device drivers are also mainly written in C, or restricted subsets of C++ providing barely no additional safety features, simply because they are tightly coupled with the kernel in all mainstream operating systems. Network device drivers managed to escape the grasp of the kernel with user space drivers such as DPDK [96] in the last years. Yet, all drivers in DPDK are written in C as large parts of them are derived from kernel implementations. DPDK consists of more than drivers: it is a whole framework for building fast packet processing apps featuring code from utility functions to complex data structures — and everything is written in C. This is not an unreasonable choice: C offers all features required for low-level systems programming and allows fine-grained control over the hardware to achieve high performance.

But with great power comes great responsibility: writing safe C code requires experience and skill. It is easy to make subtle mistakes that introduce bugs or even security vulnerabilities. Some of these bugs can be prevented by using a language that enforces certain safety properties of the program. Our research questions are: Which languages are suitable for driver development? What are the costs of safety features? A secondary goal is to simplify driver prototyping and development by providing the primitives required for user space drivers in multiple high-level languages.

We implement a user space driver tuned for performance for the Intel ixgbe family of network controllers (82599ES, X540, X550, and X552) in 9 different high-level languages featuring all major programming paradigms, memory management modes, and compilation techniques. All implementations are written from scratch in idiomatic style for the language by experienced programmers in the respective language and follow the same basic architecture, allowing for a performance comparison between the high-level languages and our reference C implementation. For most languages our driver is the first PCIe driver implementation tuned for performance enabling us to quantify the costs of different language safety features in a wide range of high-level languages. Table 4.1 lists the core properties of the selected languages.

Language	Main paradigm*	Memory management	Compilation
C	Imperative	Manual	Compiled
Rust	Imperative	Ownership/RAII†	Compiled (LLVM)
Go	Imperative	Garbage collection	Compiled
C#	Object-oriented	Garbage collection	JIT
Java	Object-oriented	Garbage collection	JIT
OCaml	Functional	Garbage collection	Compiled
Haskell	Functional	Garbage collection	Compiled (LLVM)
Swift	Protocol-oriented [1]	Reference counting	Compiled (LLVM)
JavaScript	Imperative	Garbage collection	JIT
Python	Imperative	Garbage collection	Interpreted

* All selected languages are multi-paradigm
† Ownership Based Resource Management/Resource Acquisition Is Initialization

Table 4.1: Languages used by our implementations

4.2 Background and Related work

Operating systems and device drivers predate the C programming language (1972 [145]). Operating systems before C were written in languages on an even lower level: assembly or ancient versions of ALGOL and Fortran. Even Unix started out in assembly language in 1969 before it was re-written in C in 1973 [146]. C is a high-level language compared to the previous systems: it allowed Unix to become the first portable operating system running on different architectures in 1977 [151]. The first operating systems in a language resembling a modern high-level language were the Lisp machines in the 70s. They featured operating systems written in Lisp that were fast due to hardware acceleration for high-level language constructs such as garbage collection. Both the specialized CPUs and operating systems died in the 80s [30]. Operating systems development has been mostly stuck with C since then.

Low-level packet forwarding applications traditionally deployed in the kernel due to performance requirements are moving to dedicated user-space drivers to improve performance even further. DPDK is the most feature-complete and commonly used user space driver and is free of restrictions on the choice of programming language imposed by the kernel environment. Yet, all drivers in DPDK are still written in C because large part of their implementations are taken from kernel drivers. Snabb [99] (less popular than DPDK and only 4 drivers vs. DPDK's 27 drivers) is the only other performance-optimized user space driver not written in C: It comes with drivers written in Lua running in LuaJIT [123]. However, it makes extensive use of the LuaJIT foreign function interface [124] that erodes memory safety checks that are usually present in Lua. We are not including Snabb in our performance comparison because its architecture re-

quires additional ring buffers to connect drivers and "apps", this makes it significantly slower than our drivers for the evaluated use case.

Related work on high-level languages used in domains traditionally dominated by low-level languages can be split into two categories:

1. Operating systems and unikernels in high-level languages
2. Language wrappers making low-level libraries available to applications in high-level languages

These are discussed with our implementations in the respective languages in Section 4.4.

Unrelated work: Orthogonal to our proposal is the work on XDP [82] as it does not replace drivers but adds a faster interface on top of them. Moreover, eBPF code for XDP is usually written in a subset of C [27]. P4 [21] also deserves a mention here, it is a high-level language for packet processing (but not for the driver itself). It primarily targets hardware, software implementations run on top of existing C drivers, e.g., T4P4S is using DPDK [164].

4.3 Motivation

Proponents of operating systems written in high-level languages such as Biscuit (Go) [31], Singularity (Sing#, related to C#) [65], JavaOS [147], House (Haskell) [61], and Redox (Rust) [143] tout their safety benefits over traditional implementations. Of these only Redox is under active development with the goal of becoming a production system, the others are research projects and/or abandoned. These systems are safer, but it is unlikely that the predominant desktop and server operating systems will be replaced by them any time soon.

Safe and secure operating systems can also be written in C by applying formal methods. Doing so gives even stricter guarantees than the aforementioned high-level language systems. One example of such an effort is the seL4 project, a microkernel written in C. Their kernel consists of 8,700 lines of C and 600 lines of assembly and it required an effort of 2.2 person years to build. The formal verification took another 20 person years, showing that full format verification is an enormous overhead reserved for highly specialized requirements. Their work is orthogonal to ours: seL4 relies on user space drivers which are not included these numbers. [91]

Our work builds on Linux for practicality reasons, but the concepts are independent of the underlying operating system. Given the immense effort required for full formal verification it is not likely that we will see formally verified drivers on commodity servers

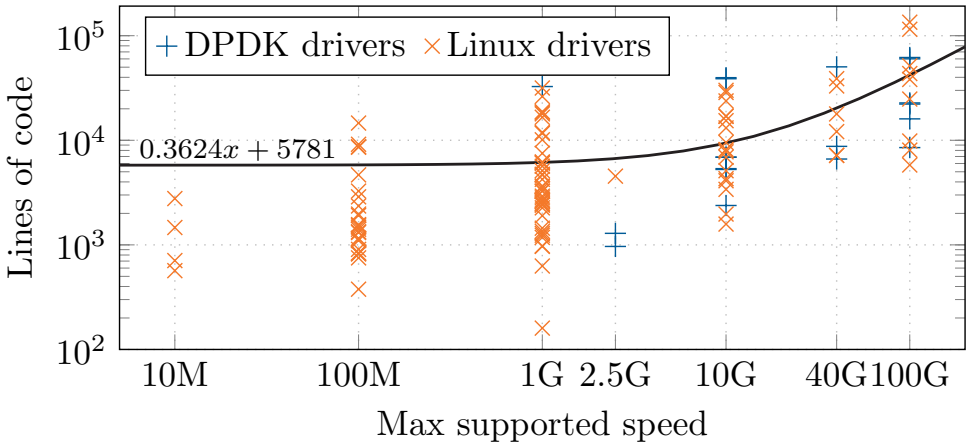


Figure 4.1: NIC technology node vs. driver size

in the near future. It is not our goal to build a perfect system, we want to reduce the attack surface (see Section 4.3.2). We argue that the goal of building a better operating system with a lower attack surface can be achieved without replacing the entire operating system. It is feasible to start writing user space drivers in safer languages today, gradually moving parts of the system to better languages.

Drivers are also the largest attack surface (by lines of code) in modern operating systems and they keep growing in complexity as more and more features are added. There are real-world security issues in drivers that could have been prevented if they were written in a high-level language [31]. User space drivers are more isolated from the rest of the system than kernel drivers and can even run without root privileges if IOMMU hardware is available, see Section 3.3.5.

4.3.1 Growing Complexity of Drivers

66% of the code in Linux 4.19 (current LTS at the time this study was done) is driver code: 11.2 M lines out of 17 M in total, 21% (2.35 M lines) of the driver code is in network drivers. 10 years earlier (in 2009), Linux 2.6.29 had only 53% of the code in drivers (3.7 M out of 6.9 M lines). Going back 10 more years to Linux 2.2.0 in 1999, we count 54% in drivers (646 k out of 1.2 M) with only 13% in network drivers. One reason for the growing total driver size is that there are more drivers to support more hardware.

Individual drivers are also increasing in complexity as hardware complexity is on the rise. Many new hardware features need support from drivers, increasing complexity and attack surface even when the number of drivers running on a given system does not change. Figure 4.1 shows a linear correlation ($R^2 = 0.3613$) between NIC technology

node and driver complexity as a log-log scatter plot. The plot considers all Ethernet drivers in Linux 4.19 and all network drivers in DPDK that do not rely on further external libraries. We also omit DPDK drivers for FPGA boards (as these expect the user to bring their own hardware and driver support), unfinished drivers (Intel ice), drivers for virtual NICs, and 4 obsolete Linux drivers for which we could not identify the speed. The linear relationship implies that driver complexity grows exponentially as network speed increases exponentially.

4.3.2 Security Bugs in Linux

Cutler et al. evaluate security bugs leading to arbitrary code execution in the Linux kernel found in 2017 [31]. They identify 65 security bugs leading to code execution and find that 8 of them are use-after-free or double-free bugs, 32 are out-of-bounds accesses, 14 are logic bugs that cannot be prevented by the programming language, and 11 are bugs where the effect of the programming language remains unclear [31]. The 40 memory bugs (61% of the 65 bugs) can be mitigated by using a memory-safe language such as their Go operating system Biscuit or our implementations. Performing an out-of-bounds access is still a bug in a memory-safe language, but it will only crash the program if it remains unhandled; effectively downgrading a code execution vulnerability to a denial of service vulnerability. User space drivers can simply be restarted after a crash, crashing kernel drivers usually take down the whole system.

We analyze these 40 memory bugs identified by Cutler et al. further and find that 39 of them are located in 11 different device drivers (the other bug is in the Bluetooth stack). The driver with the most bugs in this data set is the Qualcomm WiFi driver with 13 bugs, i.e., a total of 20% of all code execution vulnerabilities in the study could have been prevented if this network driver was written in a memory safe high-level language. The key result here is that rewriting device drivers in safer languages achieves 97.5% of the improvement they gained by rewriting the whole operating system.

In theory, eliminating bugs by moving to a high-level languages could be done without moving to the user space. In practice, the tight coupling of language and kernel makes this harder than it seems. Our proposal is to decouple drivers (which has other benefits such as isolation) and kernel to solve this.

4.3.3 Memory Safety Bugs in Windows

Microsoft's Windows kernel is written in C and C++ and hence suffers from the same preventable problem. They identified that about 70% of their security issues are due to problems with memory safety [161]. The relative percentage of memory safety bugs has been unchanged since 2006, this shows that efforts to improve memory safety in inher-

Year	DPDK*	netmap†	Snabb*	PF_RING†	PSIO†	PFQ†
2010	0/0	0/0	0/0	0/0	1/1	0/0
2011	0/0	0/0	0/0	1/1	0/0	0/0
2012	0/1	1/4	0/0	0/4	0/1	0/1
2013	0/0	3/8	0/0	0/0	0/1	0/0
2014	4/11	4/14	0/0	0/4	1/1	1/2
2015	9/15	6/14	1/2	4/7	2/3	1/2
2016	12/22	1/12	0/1	0/1	0/0	1/1
2017	17/23	3/10	0/0	1/2	1/1	1/1
Sum	41/72	19/64	1/3	6/19	5/8	4/7

* User space driver
† Kernel driver with dedicated user space API

Table 4.2: Packet processing frameworks used in academia, cells are uses/mentions; e.g., 1/3 means 3 papers mention the framework, 1 of them uses it

ently unsafe languages have not helped. Microsoft is exploring memory-safe languages for use in their operating system, starting with Rust [161].

4.3.4 The Rise of DPDK

Chapter 2 used the open source projects Click, Open vSwitch, and pfSense as motivating examples to show how network applications moved from running completely in the kernel to user space frameworks with a kernel driver (e.g., netmap) to full user space drivers (e.g., DPDK). This trend towards DPDK is also present in academia. We run a full-text search for all user space packet processing frameworks on all publications between 2010 and 2017 in the ACM conferences SIGCOMM and ANCS, and the USENIX conferences NSDI, OSDI, and ATC ($n = 1615$ papers). This yields 113 papers which we skimmed to identify whether they merely mention a framework or whether they build their application on top of it or even present the framework itself. Table 4.2 shows how DPDK started to dominate after it was open sourced in 2013[1]. The previously popular netmap is still used as a reference to explain packet processing concepts.

4.3.5 Languages Used for DPDK Applications

Applications on top of DPDK are also not restricted to C, yet most users opt to use C when building a DPDK application. This is a reasonable choice given that DPDK comes with C example code and C APIs, but there is no technical reason preventing programmers from using any other language. In fact, DPDK applications in Rust, Go, and Lua exist [127, 71, 42].

[1] The paper mentioning DPDK in 2012 is the original netmap paper [148] referring to DPDK as a commercial offering with similar goals.

DPDK's website showcases 21 projects building on DPDK. 14 (67%) of these are written in C, 5 (24%) in C++, 1 in Lua [42], and 1 in Go [71]. There are 116 projects using the #dpdk topic on GitHub, 99 of these are applications using DPDK; the others are orchestration tools or helper scripts for DPDK setups. 69 (70%) of these are written in C, 12 in C++, 4 in Rust (3 related to NetBricks [127], 1 wrapper), 4 in Lua (all related to MoonGen [42]), 3 in Go (all related to NFF-Go [71]), and one in Crystal (a wrapper).

Determining whether 67% to 70% of applications being written in C is unusually high requires a comparison baseline. Further up the network stack are web application frameworks (e.g., nodejs or jetty) that also need to provide high-speed primitives for applications on the next higher layer. Of the 43 web platforms evaluated in the TechEmpower benchmark [160] (the largest benchmark collection featuring all relevant web platforms) only 3 (7%) are written in C. 17 different languages are used here, the dominant being Java with 19 (44%) entries. Of course not all platforms measured here are suitable for applications requiring high performance. But even out of the 20 fastest benchmarked applications ("Single query" benchmark) only one is written in C and 3 in C++. Java still dominates with 7 here. Go (3), Rust (2), and C# (1) are also present in the top 20. This shows that it is possible to build fast applications on a higher layer in languages selected here.

4.3.6 User Study: Mistakes in DPDK Applications Written in C

We task students with implementing a simplified IPv4 router in C on top of DPDK for one of our networking classes. The students are a mix of undergraduate and postgraduate students with prior programming experience, a basic networking class is a pre-requisite for the class. Students are provided with a skeleton layer 2 forwarding program that handles DPDK setup, threading logic, and contains a dummy routing table. Only packet validation according to RFC 1812 [13], forwarding via a provided dummy routing table implementation, and handling ARP is required for full credits. ICMP and fragmentation handling is not required.

We received 55 submissions with at least partially working code, i.e., code that at least forwards some packets, in which we identified 3 types of common mistakes summarized in Table 4.3. Incorrect programs can contain more than one class of mistake. Logic errors are the most common and include failing to check EtherTypes, forgetting validations, and getting checksums wrong, these cannot be prevented by safer languages. Memory bugs including use-after-free bugs can be prevented by the language. No out-of-bounds access was made because the exercise offers no opportunity to do so: the code only needs to operate on the fixed-size headers which are smaller than the minimum packet size.

Total	No mistakes	Logic error	Use-after-free	Int overflow
55	12	28	13	14
100%	22%	51%	24%	25%

Table 4.3: Mistakes made by students when implementing an IPv4 router in C on top of DPDK

All integer overflow bugs happened due to code trying to decrement the time-to-live field in place in the packet struct without first checking if the field was already 0.

Ethical considerations. As handling errors done by humans requires special care we took all precautions to protect the privacy of the involved students. The study was conducted by the original corrector of the exercise, these results are used for teaching the class. No student code, answers, or any identifying information was ever given to anyone not involved in teaching the class. All submissions are pseudonymized. We were able to achieve the ethical goals of avoiding correlating errors with persons.

4.3.7 Summary

To summarize: (network) drivers are written in C which leads to preventable bugs in both the drivers (Section 4.3.2) and in applications on top of them (Section 4.3.6). Constraints imposed by the kernel environment requiring C no longer apply (Section 4.3.4). Driver complexity is growing (Section 4.3.1), so let's start using safer languages.

4.4 Implementations in High-Level Languages

All of our implementations are written from scratch by experienced programmers in idiomatic style for the respective language. We target Linux on x86 using the `uio` subsystem to map PCIe resources into user space programs which we previously explained in Section 3.3.3.

4.4.1 Architecture

All of our drivers implement the same basic architecture as the C driver discussed in the previous chapter. Our C driver serves as reference implementation that our high-level drivers are compared against. To summarize the architecture: all drivers are poll mode drivers without interrupts, all APIs are based on batches of packets, and DMA buffers are managed in custom memory pools. The memory pool for DMA buffers is also needed despite automatic memory management: not all memory is suitable for use as DMA buffers, simply using the language's allocator is not possible. This restriction potentially circumvents some of the memory safety properties of the languages in parts of the driver.

4.4.2 Challenges for High-Level Languages

There are three main challenges, all related to memory access, for user space drivers in high-level languages compared to drivers in C. Refer to Section 3.3.3 for details about how device memory is exposed to user space drivers in Linux.

Handling external memory

Two memory areas cannot be managed by the default memory allocator of the language itself: memory-mapped IO regions and DMA buffers. The former are provided by the device itself, the latter need special flags during allocation. These two requirements can be handled by a custom memory allocator which needs access to the `mmap` and `mlock` syscalls. We use a small C function in languages where these syscalls are either not available at all or only supported with restricted flags.

Unsafe primitives

External memory, i.e., PCIe address space and DMA buffers, must be wrapped in language-specific constructs that enforce bounds checks and allow access to the backing memory. Many languages come with dedicated wrapper structs that are constructed from a raw pointer and a length. For other languages we have to implement these wrappers ourselves.

In any case, all drivers need to perform inherently unsafe memory operations that cannot be checked by any language feature. However, just because some memory operations must be unsafe does not mean that we should use a language in which *all* memory operations are unsafe. The goal is to avoid unsafe operations wherever possible and restrict the remaining unsafe operations to as few places as possible to reduce the amount of code that needs to be manually audited for memory errors. In our Rust implementation we only need unsafe memory access in 13% of the code, see Section 4.4.3.

An example for an inherently unsafe memory operation is accessing device registers. No language runtime can have knowledge about how big the register file that is mapped by our custom memory allocator really is. However, access to it can be restricted to a single function that contains a bounds check.

This is, to some degree, a problem of the flat memory model of PCIe devices on the x86 platform. If registers were accessed via specialized IO instructions, we could avoid damage caused by out of bounds accesses to IO registers. x86 supports dedicated IO ports, but accessing devices registers through them is deprecated in PCIe [130].

Memory access semantics

Memory-mapped IO regions are memory addresses that are not backed by memory, each access is forwarded to a device and handled there. Simply telling the language to read or write a memory location in these regions can cause problems as optimizers make assumptions about the behavior of the memory. For example, writing a control register and never reading it back looks like a dead store to the language and the optimizer is free to remove the access. Repeatedly reading the same register in a loop while waiting for a value to be changed by the device, i.e., polling a device, looks like an opportunity to hoist the read out of the loop.

C solves this problem with the `volatile` keyword guaranteeing that at least one read or write access is performed. The high-level language needs to offer control over how these memory accesses are performed. Atomic accesses and memory barriers found in concurrency utilities make stronger guarantees and can be used instead if the language does not offer volatile semantics. Primitives from concurrency utilities can also substitute the access-exactly-once semantics required for some device drivers.

Readers interested in gory details about memory semantics for device drivers are referred to the Linux kernel documentation on memory barriers [95]. It is worth noting that all modern CPU architectures offer a memory model with cache-coherent DMA simplifying memory handling in drivers. We only test on x86 as proof of concept, but DPDK's support for x86, ARM, POWER, and Tilera shows that user space drivers themselves are possible on a wide range of modern architectures. Some of our implementations in dynamic languages likely rely on the strong memory model of x86 and might require modifications to work reliably on architectures with a weaker memory model such as ARM.

4.4.3 Rust Implementation

Rust is an obvious choice for safe user space drivers: safety and low-level features are two of its main selling points. Its ownership-based memory management allows us to use the native memory management even for our custom DMA buffers. We allocate a lightweight Rust struct for each packet that contains metadata and owns the raw memory. This struct is essentially being used as a smart pointer, i.e., it is often stack-allocated. This object is in turn either owned by the memory pool itself, the driver, or the user's application. The compiler enforces that the object has exactly one owner and that only the owner can access the object, this prevents use-after-free bugs despite using a completely custom allocator. Rust is the only language evaluated here that protects against use-after-free bugs and data races in memory buffers.

External memory is wrapped in `std::slice` objects that enforce bounds checks on each access, leaving only one place tagged as unsafe that can be the source of memory errors: we need to pass the correct length when creating the slice. Volatile memory semantics for accessing device registers are available in the `ptr` module.

IOMMU and interrupt support

We also implemented support for `vfio` in the Rust driver, all other implementations only support the simpler `uio` interface. This interface enables us to use the IOMMU to isolate the device and run without root privileges and to use interrupts instead of polling under low load. Refer to Section 3.3.3 for a detailed discussion on the `vfio` vs `uio` interfaces.

Related work

NetBricks (2016) [127] is a network function framework that allows users to build and compose network functions written in Rust. It builds on DPDK, i.e., the drivers it uses are written in C. They measure a performance penalty of 2% to 20% for Rust vs. C depending on the network function being executed. This is consistent with our results, see Section 4.6.2.

We also ported our driver to Redox (2015) [143], a real-world operating system under active development featuring a microkernel written in Rust with user space drivers. It features two network drivers for the Intel e1000 NIC family (predecessor of the ixgbe family used here) and Realtek rtl8168 NICs. Table 4.4 compares how much unsafe code their drivers use compared to our implementations. Inspecting the pre-existing Redox drivers shows several places where unsafe code could be made safe with some more work as showcased by our Redox port. Line count for our Linux driver includes all logic to make user space drivers on Linux work, our Linux version therefore has more unsafe code than the Redox version which already comes with the necessary functionality. These line counts also show the relationship between NIC speed and driver complexity hold true even for minimal drivers in other operating systems.

Driver	NIC Speed	Code [Lines]	Unsafe [Lines]	% Unsafe
Our Linux implementation	10 Gbit/s	961	125	13.0%
Our Redox implementation	10 Gbit/s	901	68	7.5%
Redox e1000	1 Gbit/s	393	140	35.6%
Redox rtl8168	1 Gbit/s	363	144	39.8%

Table 4.4: Unsafe code in different Rust drivers

4.4.4 Go Implementation

Go is a compiled systems programming language maintained by Google that is often used for distributed systems. Memory is managed by a garbage collector tuned for low latency. It achieved pause times in the low millisecond range in 2015 [144] and sub-millisecond pause times since 2016 [11].

External memory is wrapped in slices to provide bounds checks. Memory barriers and volatile semantics are indirectly provided by the atomic package which offers primitives with stronger guarantees than required.

Related work

Biscuit (2018) [31] is a research operating system written in Go that features a network driver for the same hardware as we are targeting here. Unlike all other research operating systems referenced here, they provide an explicit performance comparison with C. They observe GC pauses of up to 115 µs in their benchmarks and an overall performance of 85% to 95% of an equivalent C version. Unfortunately it does not offer a feasible way to benchmark only the driver in isolation for a comparison.

NFF-GO (2017) [71] is a network function framework allowing users to build and compose network functions in Go. It builds on DPDK, i.e., the drivers it uses are written in C. Google's Fuchsia [56] mobile operating system features a TCP stack written in Go on top of C drivers.

4.4.5 C# Implementation

C# is a versatile JIT-compiled and garbage-collected language offering both high-level features and systems programming features. Several methods for handling external memory are available, we implemented support for two of them to compare them. `Marshal` in `System.Runtime.InteropServices` offers wrappers and bounds-checked accessors for external memory. C# also offers a more direct way to work with raw memory: its unsafe mode enables language features similar to C, i.e., full support for pointers with no bounds checks and volatile memory access semantics.

Related work

The Singularity (2004) research operating system [65] is written in Sing#, a dialect of C# with added contracts and safety features developed for use in Singularity. It comes with a driver for Intel 8254x PCI NICs that are architecturally similar to the 82599 NICs used here: their DMA ring buffers are virtually identical. All memory accesses in their drivers are facilitated by safe APIs offered by the Singularity kernel. Safety is

ensured by both static verification and runtime checks to ensure that the driver cannot access memory that it is not supposed to access.

4.4.6 Java Implementation

Java is a JIT-compiled and garbage-collected language similar to C# (which was heavily inspired by Java). The only standardized way to access external memory is by calling into C using JNI, a verbose and slow foreign function interface. We target OpenJDK 12 which offers a non-standard way to handle external memory via the `sun.misc.Unsafe` object that provides functions to read and write memory with volatile access semantics. We implement and compare both methods here. Java's low-level features are inferior compared to C#, the non-standard `Unsafe` object is cumbersome to use compared to C#'s unsafe mode with full pointer support. Moreover, Java does not support unsigned integer primitives requiring work-arounds as hardware often uses such types.

Related work

JavaOS (1996) [147] was a commercial operating system targeting embedded systems and thin clients written in Java, it was discontinued in 1999. Their device driver guide [157] contains the source code of a 100 Mbit/s network driver written in Java as an example. The driver implements an interface for network drivers and calls out to helper functions and wrapper provided by JavaOS for all low-level memory operations.

4.4.7 OCaml Implementation

OCaml is a compiled functional language with garbage collection. We use OCaml `Bigarray`s backed by external memory for DMA buffers and PCIe resources, allocation is done via C helper functions. The `Cstruct` library [107] from the MirageOS project [100] allows us to access data in the arrays in a structured way by parsing definitions similar to C struct definitions and generating code for the necessary accessor functions.

Related work and MirageOS integration

We also ported our driver to MirageOS (2013) [100], a framework for creating unikernels written in OCaml with the main goal of improving security. MirageOS is not optimized for performance (e.g., all packets are copied when being passed between driver and network stack) and no performance evaluation is given by its authors (performance regression tests are being worked on [108]). MirageOS targets Xen, normal Unix processes, and a multitude of hypervisors using the Solo5 unikernel execution environment [33] including Linux' KVM. The Xen version has a driver-like interface for Xen netfront

(not a PCIe driver, though), the KVM version builds on the Solo5 unikernel execution environment [33] that provides a VirtIO driver written in C.

Our port is the first PCIe driver written in OCaml in MirageOS, we target both the unix backend and KVM backend. We added PCIe support to both MirageOS and Solo5 to enable this. Our Solo5 implementation of PCIe makes use of the IOMMU via the vfio Linux subsystem to run unprivileged and safe drivers in unikernels.

4.4.8 Haskell Implementation

Haskell is a compiled functional language with garbage collection. All necessary low-level memory access functions are available via the `Foreign` package. Memory allocation and mapping is available via `System.Posix.Memory`.

Related work

House (2005) [61] is a research operating system written in Haskell focusing on safety. Its functional interface to memory management, hardware, user-mode processes, and low-level device IO enables formal verification using P-Logic [88]. No quantitative performance evaluation is given.

PFQ [18] is a packet processing framework offering a Haskell interface and pfq-lang, a specialized domain-specific language for packet processing in Haskell. It runs on top of a kernel driver in C. Despite the focus on Haskell it is mainly written in C as it relies on C kernel modules: PFQ is 75% C, 10% C++, 7% Haskell by lines of code.

4.4.9 Swift Implementation

Swift is a compiled language maintained by Apple mainly targeted at client-side application development. Memory in Swift is managed via automatic reference counting, i.e., the runtime keeps a reference count for each object and frees the object once it is no longer in use. Despite primarily targeting end-user applications, Swift also offers all features necessary to implement drivers. Memory is wrapped in `UnsafeBufferPointer`s (and related classes) that are constructed from an address and a size. Swift only performs bounds checks in debug mode.

Related work

No other drivers or operating systems in Swift exist. The lowest level Swift projects that are available are the Vapor [163] and Kitura [90] frameworks for server-side Swift.

4.4.10 JavaScript Implementation

We build on Node.js [113] with the V8 JIT compiler and garbage collector, a common choice for server-side JavaScript. We use `ArrayBuffer`s to wrap external memory in a safe way, these arrays can then be accessed as different integer types using `TypedArray`s, circumventing JavaScript's restriction to floating point numbers. We also use the `BigInt` type that is not yet standardized but already available in Node.js. Memory allocation and physical address translation is handled via a Node.js module in C.

Related work

JavaScript is rarely used for low-level code, the most OS-like projects are NodeOS [83] and OS.js [6]. NodeOS uses the Linux kernel with Node.js as user space. OS.js runs a window manager and applications in the browser and is backed by a server running Node.js on a normal OS. Neither of these implements driver-level code in JavaScript.

4.4.11 Python Implementation

Python is an interpreted scripting language with garbage collection. Our implementation uses Cython for handling memory (77 lines of code), the remainder of the driver is written in pure Python. Performance is not the primary goal of this version of our driver, it is the only implementation presented here that is not explicitly optimized for performance. It is meant as a simple prototyping environment for PCIe drivers and as an educational tool.

Writing drivers in scripting languages allows for quick turn-around times during development or even an explorative approach to understanding hardware devices in an interactive shell. We provide primitives for PCIe driver development in Python that we hope to be helpful to others as this is the first PCIe driver in Python to our knowledge.

VirtIO driver

We also implemented a driver for virtual VirtIO [117] NICs here to make this driver accessible to users without dedicated hardware. A provided Vagrant [64] file allows spinning up a test VM to get started with PCIe driver development in Python in a safe environment.

Related work

Python is a popular [141] choice for user space USB drivers with the PyUSB library [142]. In contrast to our driver, it is mainly a wrapper for a C library. Python USB drivers are used for devices that either mainly rely on bulk transfers (handled by the underlying C library) or that do not require many transfers per second.

Lang.	Lines of code[1]	Lines of C code[1]	Source size (gzip[2])
C [44]	831	831	12.9 kB
Rust	961	0	10.4 kB
Go	1640	0	20.6 kB
C#	1266	34	13.1 kB
Java	2885	188	31.8 kB
OCaml	1177	28	12.3 kB
Haskell	1001	0	9.6 kB
Swift	1506	0	15.9 kB
JavaScript	1004	262	13.0 kB
Python	1242	(Cython) 77	14.2 kB

[1] Incl. C code, excluding empty lines and comments, counted with `cloc`
[2] Compression level 6

Table 4.5: Size of our implementations stripped down to the core feature set

4.5 Evaluation

Table 4.5 compares the code size as counted with `cloc` ignoring empty lines and comments, we also include the code size after compressing it with gzip to estimate information entropy as lines of code comparisons between different languages are not necessarily fair. We stripped features not present in all drivers (i.e., all (unit-)tests, alternate memory access implementations, VirtIO support in C and Python, IOMMU/interrupt support in C and Rust) for this evaluation. We also omit register definitions because several implementations contain automatically generated lists of > 1000 mostly unused constants for register offsets. All high-level languages require more lines than C, but the Rust, Haskell, and OCaml implementations are smaller in size as their formatting style leads to many short lines. Java and JavaScript require more C code due to boilerplate requirements of their foreign function interfaces.

Table 4.6 summarizes protections against classes of bugs available to both our driver implementations and applications built on top of them. The take-away here is that high-level languages do not necessarily increase the work-load for the implementor while gaining safety benefits. Subjectively, we have even found it easier to write driver code in high-level languages — even if more lines of code were required — after figuring out the necessary low-level details of the respective language (a one-time effort).

Bugs found in the C driver

Porting the C code to high-level languages also revealed bugs in the C implementation discussed in the previous chapter. Three bugs were discovered in the C driver:

	General memory		Packet buffers		
Lang.	**OoB**[1]	**Use after free**	**OoB**[1]	**Use after free**	**Int overflows**
C	✗	✗	✗	✗	✗
Rust	✓	✓	(✓)[2]	✓	(✓)[5]
Go	✓	✓	(✓)[2]	(✓)[4]	✗
C#	✓	✓	(✓)[2]	(✓)[4]	(✓)[5]
Java	✓	✓	(✓)[2]	(✓)[4]	✗
OCaml	✓	✓	(✓)[2]	(✓)[4]	✗
Haskell	✓	✓	(✓)[2]	(✓)[4]	(✓)[6]
Swift	✓	✓	✗[3]	(✓)[4]	✓
JavaScript	✓	✓	(✓)[2]	(✓)[4]	(✓)[6]
Python	✓	✓	(✓)[2]	(✓)[4]	(✓)[6]

[1] Out of bounds accesses
[2] Bounds enforced by wrapper, constructor in unsafe or C code
[3] Bounds only enforced in debug mode
[4] Buffers are never free'd/gc'd, only returned to a memory pool
[5] Disabled by default
[6] Uses floating point or arbitrary precision integers by default

Table 4.6: Language-level protections against classes of bugs in our drivers and the C reference code

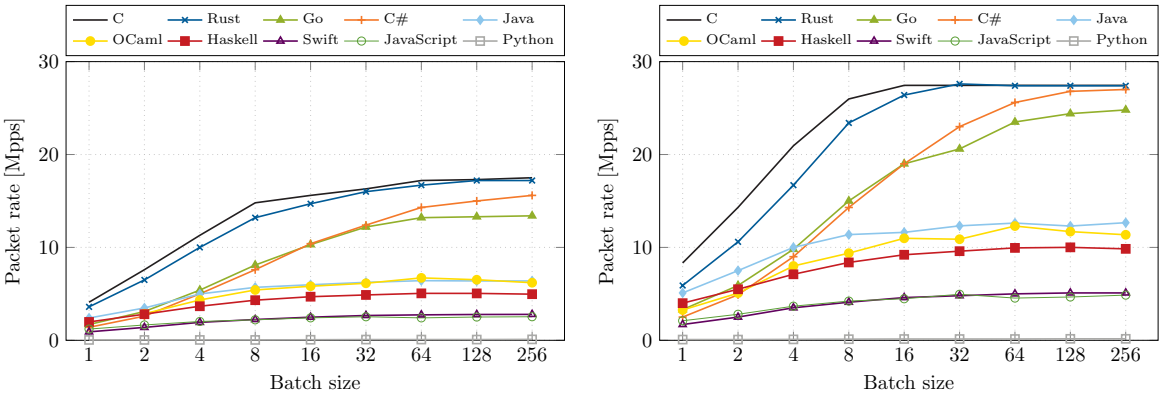
1. Memory for a queue storing configuration was allocated incorrectly, causing out-of-bounds memory accesses in certain configurations
2. Logic error in queue selection when transmitting packets
3. The VirtIO driver (not discussed in this thesis) had a bug related to buffer management in the queues

Bug 1 would be caught during run-time in a memory-safe language by bounds checks. In C it triggers undefined behavior opening doors for potential security vulnerabilities. Bug 2 was a logic error that would not have been prevented by a better language. The third bug was also unlikely to be prevented by a high-level language because it occurs in a low-level part handling pointers to DMA buffers.

So even a small driver ($\approx$ 1 000 lines) written by an experienced low-level developer (the author of this dissertation) would have 33% fewer bugs if C was a memory-safe language.

4.6 Performance

We test minimum-sized packets at full bidirectional line rate, i.e., up to 29.76 Mpps at 20 Gbit/s. Our C driver is also included in the experiments as a baseline.



(a) Forwarding rate on 1.6 GHz CPU

(b) Forwarding rate on 3.3 GHz CPU

Figure 4.2: Forwarding rate of our implementations with different batch sizes

4.6.1 Test Setup

We run our drivers on a Xeon E3-1230 v2 CPU clocked at 3.3 GHz with two 10 Gbit/s Intel X520 NICs (Server 3 from Section 2.4). Test traffic is generated with MoonGen (Chapter 5). The test application built on top of the drivers is the same as used in the previous Chapter, see Section 3.5. So all drivers configure the NICs with a queue size of 512 and run a forwarding application that modifies one byte in the packet headers. All tests are restricted to a single CPU core as we are interested in the worst-case performance (Section 2.2).

4.6.2 Effect of Batch Sizes

Processing packets in batches is a core concept of all fast network drivers (Section 3.5.3). Each received or sent batch requires synchronization with the network card, larger batches therefore improve performance. Too large batch sizes fill up the CPU's caches so there are diminishing returns or even reduced performance. 32 to 128 packets per batch is the sweet spot for user space drivers [52, 89, 14].

Figure 4.2 shows the maximum achievable bidirectional forwarding performance of our implementations. We also run the benchmark at a reduced CPU frequency of 1.6 GHz as the performance of the C and Rust forwarders quickly hit some hardware bottleneck at 94% line rate at 3.3 GHz. A few trade-offs for the conflicting goals of writing idiomatic safe code and achieving a high-performance were evaluated for each language. Haskell and OCaml allocate a new list/array for each processed batch of packets while all other languages re-use arrays. Recycling arrays in these functional languages building on immutable data structures would not be idiomatic code, this is one of the reasons for their lower performance.

Rust

Rust achieves 90% (batch size ≤ 32) to 98% (batch sizes ≥ 64) of the baseline C performance when the CPU is at 1.6 GHz[1], making it the only competitor with a comparable performance to C.

Our Redox port of the Rust driver only achieves 0.12 Mpps due to performance bottlenecks in Redox (high performance networking is not yet a goal of Redox). This is still a factor 150 improvement over the pre-existing Redox drivers due to support for asynchronous packet transmission.

Go

Go also fares reasonably well for a garbage-collected language, achieving about 76% of C's performance. This proves Go as a suitable candidate for a systems programming language if performance is not the primary goal.

C#

The aforementioned utility functions from the `Marshal` class to handle memory proved to be too slow to achieve competitive performance. Rewriting the driver to use C# unsafe blocks and raw pointers in selected places improved performance by 40%. Synthetic benchmarks show a 50-60% overhead for the safer alternatives over raw pointers. C# performs slightly better than Go with large batch sizes.

Java

Our implementation heavily relies on devirtualization, inlining, and dead-code elimination by the optimizer as it features several abstraction layers and indirections that are typical for idiomatic Java code. We used profiling to validate that all relevant optimization steps are performed, almost all CPU time is spent in the transmit and receive functions showing up as leaf functions despite calling into an abstraction layer for memory access.

We use OpenJDK 12 with the HotSpot JIT, the *Parallel* garbage collector, and the Unsafe object for memory access. Using JNI for memory access instead is several orders of magnitude slower. Using OpenJ9 as JIT reduces performance by 14%. These results are significantly slower than C# and show that C#'s low-level features are crucial for fast drivers. One reason is that C# features *value types* that avoid unnecessary heap allocations. We try to avoid object allocations by recycling packet buffers, but

[1] To avoid hitting unknown hardware bottlenecks observed at 3.3 GHz

Batch\GC	CMS	Serial	Parallel	G1	ZGC	Shenandoah	Epsilon
4	9.8	10.0	10.0	7.8	9.4	8.5	10.0
32	12.3	12.4	12.3	9.3	11.5	10.8	12.2
256	12.6	12.4	12.3	9.7	11.6	11.2	12.7

Table 4.7: Performance of different Java garbage collectors in Mpps when forwarding packets at 3.3 GHz

writing allocation-free code in idiomatic Java is virtually impossible, so there are some unavoidable allocations. OpenJDK 12 features 7 different garbage collectors which have an impact on performance as shown in Table 4.7. Epsilon is a no-op implementation that never frees memory, leaking approximately 20 bytes per forwarded packet. Note that simply leaking memory is not necessarily faster than properly free'ing it, re-using memory can be cheaper than allocating new one due to improved locality and fewer requests to the OS for more and more memory.

OCaml

Enabling the Flambda [118] optimizations in OCaml 4.07.0 increases throughput by 9%. An interesting optimization is representing bit fields as separate 16 bit integers instead of 32 bit integers if possible: Larger integers are boxed[1] in the OCaml runtime. This increases performance by 0.7% when applied to the status bits in the DMA descriptors.

Our MirageOS port achieves 3.25 Gbit/s TCP throughput using iperf and the Mirage TCP stack on Mirage's Unix backend (i.e., running Mirage as a process, not as a unikernel in a VM). Targeting KVM (and enabling the IOMMU to do so) yields the same throughput. This is an improvement of more than 100% over the built-in networking in the KVM backend that exposes a TAP network device and achieves 1.6 Gbit/s in the same test.

Haskell

Compiler (GHC 8.4.3) optimizations seem to do more harm than good in this workload. Increasing the optimization level in the default GHC backend from `O1` to `O2` reduces throughput by 11%, we could not identify any reason for this. The data in the graph is based on the LLVM backend which is 3.5% faster than the default backend at `O1`. Enabling the threaded runtime in GHC decreases performance by 8% and causes the driver to lose packets even at loads below 1 Mpps due to regular GC pauses of several milliseconds.

[1] Meaning they require additional metadata for the garbage collector.

Swift

Swift increments a reference counter for each object passed into a function and decreases it when leaving the function. This is done for every single packet as they are wrapped in Swift-native wrappers for bounds checks. There is no good way to disable this behavior for the wrapper objects while maintaining an idiomatic API for applications using the driver. A total of 76% of the CPU time is spent incrementing and decrementing reference counters. This is the only language runtime evaluated here that incurs a large cost even for objects that are never free'd.

JavaScript

We also compare Node.js versions 10 (V8 6.9, current LTS), 11 (V8 7.0), and 12 (V8 7.5), older versions are unsupported due to lack of `BigInt` support. Node 10 and 11 perform virtually identical, upgrading to 12 degrades performance by 13% as access to `TypedArray`s is slower in this version.

The main problem is the lack of normal integer data types in JavaScript, the only number types available are 64 bit floating point values (doubles) and integers of variable lengths called `BigInts`. Doubles have sufficient precision to store up to 54 bit long integers, so it is possible to store many values in them. `TypedArray`s (which are faster than plain `DataView`s) can be used to read up to 32 bit from raw memory into double values which can be used to avoid `BigInts` in many places. However, sometimes 64 bit values are unavoidable in which case `BigInts` are required which are slower than doubles. Especially constructing `BigInts` proved to be a performance bottleneck.

Python

Python only achieves 0.14 Mpps in the best case using the default CPython interpreter in version 3.7. Most time is spent in code related to making C structs available to higher layers. We are using constructs that are incompatible with the JIT compiler PyPy. This is the only implementation here not optimized for performance, we believe it is possible to increase throughput by an order of magnitude by re-cycling struct definitions. Despite this we are content with the Python implementation as the main goal was not a high throughput but a simple implementation in a scripting language for prototyping functionality and to demonstrate implementation equivalence.

4.6.3 The Cost of Safety Features in Rust

Rust is our fastest implementation. It is also the only high-level language without overhead for memory management here, making it an ideal candidate to investigate

	Batch 32, 1.6 GHz		Batch 8, 1.6 GHz	
Events per packet	**C**	**Rust**	**C**	**Rust**
Cycles	94	100	108	120
Instructions	127	209	139	232
Instr. per cycle	1.35	2.09	1.29	1.93
Branches	18	24	19	27
Branch mispredicts	0.05	0.08	0.02	0.06
Store µops	21.8	37.4	24.4	43.0
Load µops	30.1	77.0	33.4	84.2
Load L1 hits	24.3	75.9	28.8	83.1
Load L2 hits	1.1	0.05	1.2	0.1
Load L3 hits	0.9	0.0	0.5	0.0
Load L3 misses	0.3	0.1	0.3	0.3

Table 4.8: Performance counter readings in events per packet when forwarding packets

overhead further by profiling. There are only two major differences between the Rust and C implementations:

(1) Rust enforces bounds checks while our C code contains no bounds checks (arguably idiomatic style for C).

(2) C does not require a wrapper object for DMA buffers, it stores all necessary metadata directly in front of the packet data the same memory area as the DMA buffer.

However, the Rust wrapper objects can be stack-allocated and effectively replace the pointer used by C with a smart pointer, mitigating the locality penalty. The main performance disadvantage is therefore bounds checking.

We use CPU performance counters (read by `perf stat` over a period of 30 seconds under full bidirectional load) to profile our forwarder with two different batch sizes. Table 4.8 lists the results in events per forwarded packet. Recall the key performance numbers from Section 2.2: we have up to 222 cycles available in total per packet on a 3.3 GHz CPU to achieve our performance goal of 14.88 Mpps on a single core.

Rust requires 65% (67%) more instructions to forward a single packet at a batch size of 32 (8). The number of branches executed rises by 33% (42%), the number of loads even by 150% (180%). However, the Rust code only requires 6 (12) more cycles per packet overall despite executing 82 (93) more instructions per packet. Synthetic benchmarks can achieve an even lower overhead of bounds checking [45].

A modern superscalar out of order processor can effectively hide the overhead introduced by these safety checks: normal execution does not trigger bounds check violations, the processor is therefore able to correctly predict (branch mispredict rate is at 0.2% - 0.3%)

and speculatively execute the correct path. The Rust code achieves about 2 instructions per cycle vs. about 1.3 instructions with the C code.

A good example of speculatively executing code in the presence of bounds checks is the Spectre v1 security vulnerability which exists due to this performance optimization in CPUs [139]. Note that user space drivers are not affected by this vulnerability as the program's control flow does not cross a trust boundary (kernel space vs. user space) as everything runs in the same process.

Caches also help with the additional required loads of bounds information: this workload achieves an L1 cache hit rate of 98.5% (98.7%). Note that the sum of cache hits and L3 misses is not equal to the number of load µops because some loads are executed immediately after the store, fetching the data from the store buffer before it even reaches the cache.

Another safety feature in Rust are integer overflow checks that catch mistakes common in our user study (Section 4.3.6). Overflow checks are currently disabled by default in release mode in Rust and have to be explicitly enabled with a compile-time flag. Doing so decreases throughput by only 0.8% at batch size 8, the change with larger batch sizes was less than 0.1%. Profiling shows that 9 additional instructions per packet are executed by the CPU, 8 of them are branches. Total branch mispredictions are unaffected, i.e., the branch check is always predicted correctly by the CPU. This is another instance of speculative execution in an out-of-order CPU hiding the cost of safety features.

4.6.4 Comparison with Other Language Benchmarks

Table 4.9 compares our performance results with the Computer Language Benchmarks Game (CLBG) [60], a popular more general performance comparison of different languages. We use the "fastest measurement at the largest workload" data set from 2019-07-21, summarized as geometric mean [46] over all benchmarks. Our results are for batch size 32 (realistic value for real-world applications, e.g., DPDK default) at 1.6 GHz CPU speed to enforce a CPU bottleneck.

This shows that especially dynamic and functional languages pay a performance penalty when being used for low-level code compared to the more general benchmark results.

Benchmark\Lang.	**Rust**	**Go**	**C#**	**Java**	**OCaml**	**Haskell**	**Swift**	**JS**	**Py.**
Our results	98%	81%	76%	38%	38%	30%	16%	16%	1%
CLBG [60]	117%	34%	73%	52%	80%	65%	64%	28%	4%

Table 4.9: Performance results normalized to C, i.e., 50% means it achieves half the speed of C

Our implementations (except Python) went through several rounds of optimization based on profiling results and extensive benchmarks. While there are certainly some missed opportunities for further minor optimization, we believe to be close to the optimum achievable performance for drivers in idiomatic code in these languages. Note that the reference benchmark is also probably not perfect. Go and C# perform even better at the low-level task of writing drivers than the general purpose benchmarks, showing how a language's performance characteristics depend on the use case. General-purpose benchmark results can be misleading when writing low-level code in high-level languages.

Note that Go seems to be doing particularly poorly on the CLBG benchmark. The Go FAQ [57] blames this outcome on poorly optimized libraries used in these benchmarks.

4.7 Latency

Latency is dominated by the time a packet spends in buffers (due to queuing), not by time spent handling a packet on the CPU. Our drivers forward packets in hundreds of cycles, i.e., within hundreds of nanoseconds. A driver with a lower throughput is therefore not automatically one with a higher latency while operating below its maximum service rate. The main factors driving up latency are pauses due to garbage collection and the batch size.

Note that the batch size parameter is only the maximum batch size, a driver operating below its limit will process smaller batches. Drivers running closer to their limits will handle larger batches and incur a larger latency. Load is generated offered as constant bit-rate traffic (cf. RFC 2544, RFC 1242 [24, 25]), meaning a constant gap between packets, this leads to a uniform distribution of packet latencies within a batch: The first packet having the largest (it arrived just after the previous batch was processed) and the last the shortest (it arrived just before the batch was processed).

Our drivers run with ring sizes of 512 by default and configure the NIC to drop packets if the receive ring is full instead of buffering them in the NIC to avoid buffer bloat under overload conditions. More details on the effect of the ring size on latency can be found in Section 3.5.7.

4.7.1 Test Setup

Choice of language does not affect the average or median latency significantly: garbage collection pauses and JIT compilation are visible in tail latency. We therefore measure the latency of all forwarded packets by inserting fiber optic splitters on both sides of

the device under test, see Section 2.4.1 for a detailed description of the test setup. The device under test uses an Intel Xeon E5-2620 v3 at 2.40 GHz and a dual-port Intel X520 NIC (Server 1 from Section 2.4). All latencies were measured with a batch size of 32 and ring size 512 under bidirectional load with constant bit-rate traffic. The device under test has a maximum buffer capacity of 1,088 packets in this configuration. Different batch sizes, ring sizes, and NUMA configurations as discussed in the previous chapter affect latency in the same way for all programming languages and are therefore not evaluated separately for all languages.

4.7.2 Tail Latencies

Figure 4.3 shows latencies of our drivers when forwarding packets at different rates, all graphs are based on 10 million measured latencies. Note: for 20 Mpps bidirectional traffic must be used and our test setup (Section 2.4.1) only captures timestamps in one direction. The data is plotted as CCDF to focus on the worst-case latencies. Implementations not able to cope with the offered load are omitted from the graphs — their latency is simply a function of the buffer size as the receive buffers fill up completely. No maximum observed latency is significantly different from the 99.9999th percentile. Java and JavaScript lose packets during startup due to JIT compilation, we therefore exclude the first 5 seconds of the test runs for these two languages. All other tests shown ran without packet loss.

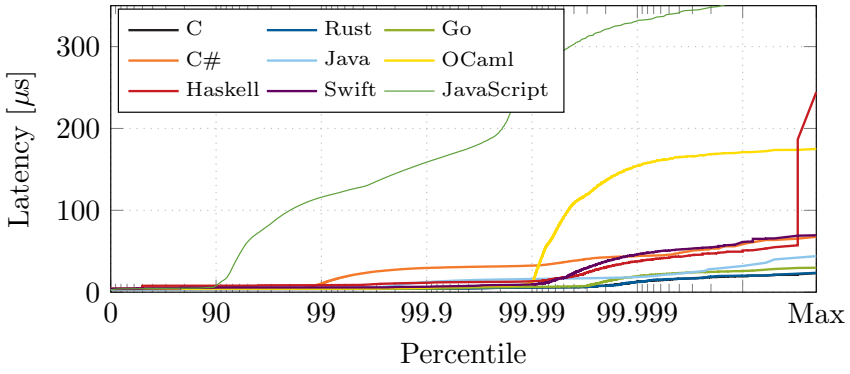
Rust and C

Even Rust and C show a skewed latency distribution with some packets taking 5 times as long as the median packet. One reason for this is that all our drivers handle periodic (1 Hz) printing of throughput statistics in the main thread. Note that the 99.9999th percentile means that one in a million packets is affected. Printing statistics once per second at 1 Mpps or more thus affects latency at this level. A second reason is that it is not possible to isolate a core completely from the system on Linux. Some very short local timer interrupts are even present with the `isolcpus` kernel option.

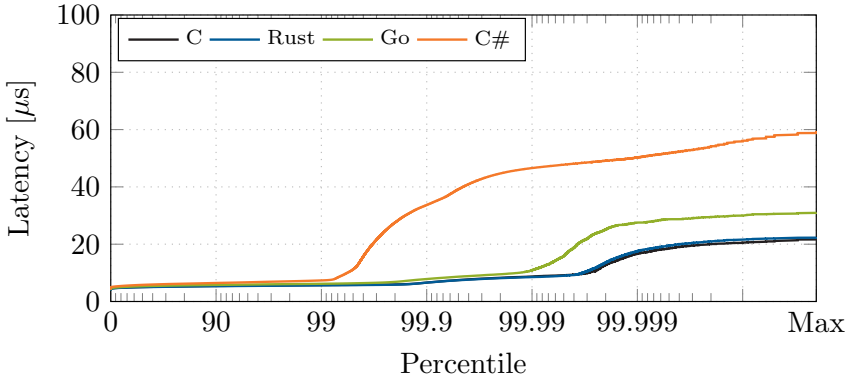
C outperforms Rust at 20 Mpps in the long tail because of non-linear effects of buffer usage and effective batch sizes as you approach the limit of the system. Even small differences in performance can result in comparatively large changes to the worst-case latency (here: difference starts to show at the $\approx$ 99.9th percentile).

Go

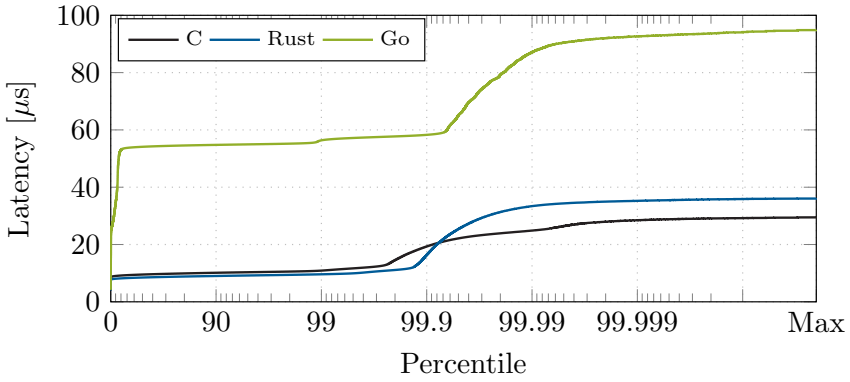
Go's low-latency garbage collector achieves the lowest pause times of any garbage-collected language here. Latency suffers at 20 Mpps because the driver operates at



(a) Forwarding latency at 1 Mpps



(b) Forwarding latency at 10 Mpps



(c) Forwarding latency at 20 Mpps

Figure 4.3: Tail latency of our implementations when forwarding packets

its limit on this system here. Cutler et al. measured a maximum garbage collection pause of 115 µs in their Go operating system, demonstrating that sub-millisecond pauses with Go are possible even in larger applications.

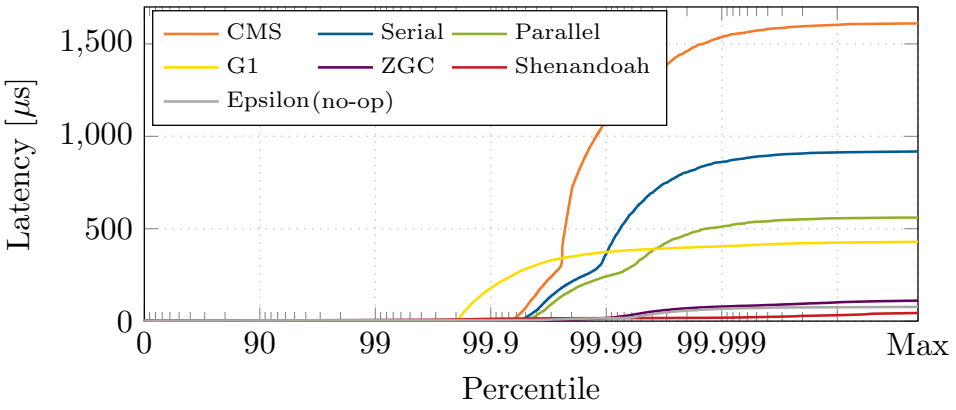


Figure 4.4: Forwarding latency of Java at 1 Mpps with different garbage collectors

C#

C# features several garbage collector modes tuned for different workloads [105]. The default garbage collector caused a latency of 240 µs at the 99.9999th percentile at 10 Mpps. Switching it to `SustainedLowLatency` reduces latency to only 55 µs, this change also reduces the maximum achievable throughput by 1.2%. All measurements were performed with the `SustainedLowLatency` garbage collector.

C# also uses a JIT compiler that might introduce additional pause times. However, the compilation of most functions happens immediately after startup even if no packets are forwarded: We implement a poll-mode driver that effectively warms up the JIT compiler.

Java

Java exhibits packet loss and excessive latencies during the first few seconds of all test runs, this is likely due to JIT compilation hitting a function only after the traffic flow starts. All latency measurements for Java therefore exclude the first 5 seconds of the test runs. Achieving these latencies with Java also required disabling periodic printing of forwarding statistics.

Figure 4.3a shows results for the Shenandoah garbage collector which exhibited the lowest latency. We also tried the different settings in Shenandoah that are recommended for low-latency requirements [120]. Neither using a fixed-size heap with pre-touched pages, nor disabling biased locking made a measurable difference. Changing the heuristic from the default *adaptive* to *static* reduces worst-case latency from 338 µs to 323 µs, setting it to *compact* increases latency to 800 µs.

Figure 4.4 compares the latency incurred by the different available garbage collectors in OpenJDK 12 while forwarding 1 Mpps. We configured the lowest possible target pause time of 1 ms. Note that the maximum buffer time with this configuration is ≈1.1 ms, i.e., the CMS collector drops packets at this rate. This could be mitigated by larger rings or by enabling buffering on the NIC if the ring is full. There is a clear trade-off between throughput and latency for the different garbage collectors, cf. Table 4.7. ZGC hits a sweet spot between high throughput and low latency. Even Epsilon (no-op, never frees objects) is also not ideal, indicating that the garbage collector is not the only cause of latency. This can likely be attributed to bad data locality as it fills up the whole address space and caches, never re-using memory.

OCaml and Haskell

Both OCaml and Haskell ship with only a relatively simple garbage collector (compared to Go, C#, and Java) not optimized for sub-millisecond pause times. Haskell even drops packets due to garbage collection pauses when the multi-threaded runtime is enabled, the single threaded runtime performs reasonably well.

Swift

It remains unclear why Swift performs worse than some garbage-collected languages. Its reference counting memory management should distribute the work evenly and not lead to spikes, but we observe tail latency comparable to the garbage-collected Haskell driver.

JavaScript

JavaScript loses packets during startup, indicating that the JIT compiler is to blame, the graph excludes the first 5 seconds. Even in the steady state the latency is far worse than the others starting at the 90th percentile. The worst-case latency is at 359 µs.

Python

Python exhibits packet loss even at low rates and is therefore excluded here, worst-case latencies are several milliseconds even when running at 0.1 Mpps.

4.8 Conclusion

Rewriting the entire operating system in a high-level language is a laudable effort but unlikely to disrupt the big mainstream desktop and server operating systems in the near future. We propose to start rewriting drivers as user space drivers in high-level languages instead as they present the largest attack surface in an operating system (Section 4.3.1).

39 of the 40 memory safety bugs in Linux examined here are located in drivers, showing that most of the security improvements can be gained without replacing the whole operating system, see Section 4.3.2. Network drivers are a good starting point for this effort: User space network drivers written in C are already commonplace. Moreover, they are critical for security: they are exposed to the external world or serve as a barrier isolating untrusted virtual machines (e.g., CVE-2018-1059 in DPDK allowed VMs to extract host memory due to a missing bounds check [37]).

Higher layers of the network stack are also already moving towards high-level languages (e.g., the TCP stack in Fuchsia [56] is written in Go) and towards user space implementations. The transport protocol QUIC is only available as user space libraries, e.g., in Chromium [58] or CloudFlare's quiche written in Rust [28]. Apple runs a user space TCP stack on mobile devices [26]. User space stacks also call for a more modern interface than POSIX sockets: the socket replacement TAPS is currently being standardized, it explicitly targets "modern platforms and programming languages" [159]. This trend simplifies replacing the kernel C drivers with user space drivers in high-level languages as legacy interfaces are being deprecated.

Addressing research question Q2 from Section 1.2 (Can high-level languages be used in software-based network devices?), we can now clearly say: yes. Our evaluation shows that Rust is a prime candidate for safer drivers: Its ownership-based management system prevents memory bugs even in custom memory areas not allocated by the language runtime, see Section 4.5.

The cost of these safety and security features are only 2% of throughput (Section 4.6.2). Rust's ownership based memory management provides more safety features than languages based on garbage collection here and it does so without affecting latency (Section 4.7.2). For these 2% in throughput we gain memory safety in 87% of the code (Section 4.4.3).

We conclude that it is worthwhile to build packet processing systems in high-level languages, especially Rust. Our Rust implementation manages to forward 27.4 Mpps on a single 3.3 GHz CPU core, well above our target of 14.88 Mpps (Section 2.2). Go and C# also manage to stay above the target rate at high CPU speeds and/or large batch sizes, but with lower margins, see graphs in Section 4.6.2.

However, this was only the lowest software level on a stack of complexity. We evaluated an application that does nothing but touch and forward a packet, having a large performance margin is important to leave processing time for an application that does something useful with the packet. Fully answering if and and how high-level languages

can be used requires us to build a real-world application in a high-level language (Chapter 5).

4.9 Author's Contributions

Sections 4.1 to 4.8 are based on the following publication [43]:

> Paul Emmerich, Simon Ellmann, Fabian Bonk, Alex Egger, Esaú García Sánchez-Torija, Thomas Günzel, Sebastian Di Luzio, Alexandru Obada, Maximilian Stadlmeier, Sebastian Voit, and Georg Carle. "The Case for Writing Network Drivers in High-Level Programming Languages". In: *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2019)*. IEEE. Cambridge, UK, Sept. 2019

All high-level drivers discussed here were implemented by a team of students (as various types of theses) advised by the author. All experiments in the paper and in this thesis are either new or re-runs. No data beside the code artifacts from the student theses were used for either the paper or this thesis.

The core architecture of all drivers is based on the author's design. All experiments and benchmarks presented here were conceived and fully specified by the author. The author also made contributions to the executions of the experiments and test runs. All analyses and conclusions drawn from the experimental results are the author's work. All comparisons between the different languages were done by the author. The micro-architectural comparison and benchmark between the C and Rust implementation was done by the author. The comparison of latency of different Java garbage collectors was done by the author. The investigation of security issues in drivers, complexity of drivers, the study about bugs in student C code, and the study of languages used for networking applications were done by the author.

The following significant changes vs. the version published in the conference proceedings[1] were made for this thesis:

- The Java driver has been optimized for lower latency, the latency measurement of it was repeated and the text was updated accordingly.
- The discussion about bugs in operating systems has been extended to include Microsoft's Windows operating system.

[1] The updated Java performance benchmarks have also been distributed on GitHub and in an updated paper on the author's web site

- A retrospective look at bugs found in the C driver while porting it to high-level languages has been added.
- The OCaml driver has been extended to be better integrated with MirageOS with IOMMU support.

Part III

Flexible Testing of Network Devices

Chapter 5

MoonGen: A fast and flexible packet generator

When we started to work on this dissertation project in 2014 the world of software-based networking research was lacking a packet generator meeting the demands of modern applications being developed. So we build our own software packet generator that applies the core ideas of our thesis: Using high-level languages in domains traditionally dominated by C or hardware implementations to gain flexibility.

MoonGen was explicitly designed to be more than just a tool required for our work: We took care to make it user friendly and accessible to the whole research community. Since its publication it has become the de-facto standard for packet generators in the community. There are over 300 papers citing MoonGen, including several high-impact papers published at SIGCOMM using MoonGen for their experimental evaluation [152, 93, 168].

We take a look at how software-based packet processing applications can be benchmarked (research question Q3 from Section 1.2). Packet generators for such applications require an extraordinarily high flexibility because testing entirely new kinds of devices requires entirely new kinds of testing protocols. MoonGen also serves as a real-world packet processing application that uses the high-level programming language Lua to gain unprecedented flexibility by running user-defined code for every packet sent out (Section 5.4). MoonGen achieves our performance goals of 14.88 Mpps on a single CPU core (Section 2.2) despite being written in a high-level scripting language.

Moreover, we take a look at replacing hardware with software and the trade-offs required for precision (research question Q4). Packet generators are traditionally a domain dom-

inated by specialized hardware, we instead use a high-level language here wherever we can. This comes with trade-offs, especially in the domains related to latency measurements. We find that we can use hardware features commonly found on commodity NIC to implement precise (±12.8 ns) timestamping in hardware (Section 5.6).

The remainder of this chapter is based on our publication about MoonGen which is joint work by Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle [42]. A full account of the author's contributions is given in Section 5.10.

5.1 Introduction

Tools for traffic generation are essential to quantitative evaluations of network performance. Hardware-based solutions for packet generation are expensive and in many cases inflexible. Existing software solutions often lack performance or flexibility and come with precision problems [23] (discussed in detail in Chapter 6).

The state of the art in packet generation, discussed further in Section 5.2, motivated us to design MoonGen. Our novel software packet generator is flexible, fast, and precise without relying on special-purpose hardware. Moving the packet generation logic into user-controlled Lua scripts ensures flexibility by utilizing modern high-level languages. We build on the JIT compiler LuaJIT [123] and the packet processing framework DPDK [96]. Our architecture and its implementation are described in detail in Section 5.3.

MoonGen is controlled through its API instead of configuration files. We explain the interface in Section 5.4 by presenting code examples for typical use cases. The API allows for applications beyond packet generation as it makes DPDK packet processing facilities available to Lua scripts. Section 5.5 evaluates the performance of our approach. We show that running Lua code for each packet is feasible and can even be faster than an implementation written in C.

Our packet generator can also receive packets and measure round-trip latencies with submicrosecond precision and accuracy. We achieve this by using the hardware features of Intel commodity NICs intended for clock synchronization across networks. Section 5.6 features a detailed evaluation of timestamping capabilities of modern commodity NICs.

Section 5.7 discusses different ways for rate control, both in software and hardware. We also present a novel method for controlling inter-packet gaps to generate well-defined traffic patterns. Our new approach requires only minimal hardware support and is superior to existing pure software implementations.

5.2 Related Work

Packet generators face a tradeoff between complexity and performance. This is reflected in the available packet generators: Barebone high-speed packet generators with limited capabilities on the one hand and feature-rich packet generators that do not scale to high data rates on the other hand. While high-speed packet generators often only send out pre-crafted Ethernet frames (e.g., pcap files), more advanced packet generators are able to transmit complex load patterns by implementing and responding to higher-layer protocols (e.g., web server load tester). Consequently, there is a lack of fast *and* flexible packet generators. Besides mere traffic generation, many packet generators also offer the possibility to capture incoming traffic and relate the generated to the received traffic.

The traditional approach to measure the performance of network devices uses hardware solutions to achieve high packet rates and high accuracy [23]. Especially their ability to accurately control the sending rate and precise timestamping are important in these scenarios. Common hardware packet generators manufactured by IXIA, Spirent, or XENA are tailored to special use cases such as performing RFC 2544 compliant device tests [25]. They send predefined traces of higher-layer protocols, but avoid complex hardware implementations of protocols. Therefore, these hardware appliances are on the fast-but-simple end of the spectrum of packet generators. They are focused on well-defined and reproducible performance tests for comparison of networking devices via synthetic traffic. However, the high costs severely limit their usage [23].

NetFPGA is an open source FPGA-based NIC that can be used as a packet generator [110]. Although costs are still beyond commodity hardware costs, it is used more often in academic publications. OSNT [7] is a packet generator and general-purpose test tool that uses it to measure latencies with nanosecond accuracy.

Software packet generators running on commodity hardware are widespread for different use cases. Especially traffic generators that emulate realistic traffic, e.g., Harpoon [155], suffer from poor performance on modern 10 Gbit/s links. We focus on high-speed traffic generators that are able to saturate 10 Gbit/s links with minimum-sized packets, i.e., achieve a rate of 14.88 Mpps. Bonelli et al. [19] implement a software traffic generator, which is able to send 12 Mpps by using multiple CPU cores. Software packet generators often rely on frameworks for efficient packet transmission [115, 148, 96] to increase the performance further to the line rate limit. Less complex packet generators can be found as example applications for high-speed packet IO frameworks: zsend for PF_RING ZC [115] and pktgen for netmap [148]. Wind River Systems provides `pktgen-dpdk` [165] for DPDK [96]. `pktgen-dpdk` features a Lua scripting API that can be used to control the parameters of the generator, but the scripts cannot modify

the packets themselves. Further, existing tools for packet generation like Ostinato have been ported to DPDK to improve their performance [122]. Previous studies showed that software solutions are not able to precisely control the inter-packet delays [23, 29]. This leads to micro-bursts and jitter, a fact that impacts the reproducibility and validity of tests that rely on a precise definition of the generated traffic.

Ostinato is the most flexible software packet solution of the investigated options as it features configuration through Python scripts while using DPDK for high-speed packet IO. However, its scripting API is limited to the configuration of predefined settings, the scripts cannot be executed for each packet. Precise timestamping and rate control are also not supported. [122]

One has to make a choice between flexibility (software packet generators) and precision (hardware packet generators) with the available options. Today different measurement setups therefore require different packet generators. For example, precise latency measurements currently require hardware solutions. Complex packet generation (e.g., testing advanced features of network middleboxes like firewalls) requires flexible software solutions. We present a hybrid solution with the goal to be usable in all scenarios.

5.3 Implementation

We identified the following requirements based on our goal to close the gap between software and hardware solutions by combining the advantages of both. MoonGen must...

(R1) ...be implemented in software and run on commodity hardware.

(R2) ...be able to saturate multiple 10 Gbit/s links with minimum-sized packets.

(R3) ...be as flexible as possible.

(R4) ...offer precise and accurate timestamping and rate control.

The following building blocks were chosen based on these requirements.

5.3.1 Packet Processing with DPDK

We chose DPDK for MoonGen because of its speed [53] and support for NICs from virtually all vendors of server-grade NICs. DPDK is also a full user space driver as discussed in Chapter 3, an important property for MoonGen: we need to modify the driver to implement requirement (R4). Another option that was considered but discarded was netmap. We implemented a prototype of MoonGen on netmap to compare it, but faced many challenges including poor support for NICs, lack of support for hardware features and worse performance than DPDK.

5.3.2 Scripting with LuaJIT

MoonGen must be as flexible as possible (R3). Therefore, we move the whole packet generation logic into user-defined scripts as this ensures the maximum possible flexibility. At the same time, the performance of the scripting engine must not inhibit the overall performance of MoonGen (R2).

We considered two scripting languages with fast implementations: Lua with LuaJIT and JavaScript with V8. There were two main reasons why LuaJIT was chosen: Related work shows that it is suitable for high-speed packet processing tasks [99] at high packet rates (R2). Moreover, its foreign function interface is excellent for an easy integration of C libraries like DPDK. V8 requires a large amount of boiler plate code and has unclear performance characteristics when it comes to converting between different data types. LuaJIT simply requires a function definition and call it with no additional overhead as data layout can be controlled to be binary-compatible with the C data types required by the library.

One concern for scripting languages is always latency due to garbage collection and compilation of code during run time as discussed in Chapter 4. Note that latency is less of a problem for packet generators than it is for general-purpose packet processing systems. MoonGen's goal is to always fill all buffers completely to handle such interruptions. This is contrary to goals of a general-purpose system that wants to avoid buffering to avoid latency.

The currently supported NICs feature buffer sizes in the order of hundreds of kilobytes [72, 73, 76]. For example, the smallest buffer on the X540 chip is the 160 kB transmit buffer, which can store 128 μs of data at 10 Gbit/s. In addition, there are DMA buffers in main memory of thousands of packets This effectively conceals short pause times if the buffer is full. Pause times introduced by the JIT compiler are in the range of "a couple of microseconds" [125]. The garbage collector (GC) works in incremental steps, the pause times depend on the usage. In practice, we find that we encounter small problems during startup before the queues are filled, we observe pause times of up to 36 µs between batches for the first 10 batches at rates of 4 Mpps and above (Section 6.6.4). All following batches are unaffected. We could not observe any adverse effects of this behavior in any benchmarks.

Lua was not one of the high-level languages evaluated in Chapter 4. The attentive reader has also noticed that the publication of MoonGen predates the research on high-level languages. Our deep dive into performance of drivers and high-level languages would simply not have been possible without having a reliable packet generator first. Lua was not included in the study of high-level languages in Chapter 4 because Lua was already

proven to work well in practice: high-speed drivers in Lua exist [99] and MoonGen is fast and widely used, so no further research was needed.

5.3.3 Hardware Architecture

Understanding how the underlying hardware works is important for the design of a high-speed packet generator. The typical operating system socket API hides important aspects of networking hardware that are crucial for the design of low-level packet processing tools.

A central feature of modern commodity NICs is support for multi-core CPUs. Each NIC supported by DPDK features multiple receive and transmit queues per network interface. This is not visible from the socket API of the operating system as it is handled by the driver [70]. For example, both the X540 and 82599 10 Gbit/s NICs support 128 receive and transmit queues. Such a queue is essentially a virtual interface and they can be used independently from each other. [73, 76]

Multiple transmit queues allow for perfect multi-core scaling of packet generation, see Section 2.1. Each configured queue can be assigned to a single CPU core in a multi-core packet generator. Receive queues are also statically assigned to threads and the incoming traffic is distributed via configurable filters (e.g., Intel Flow Director) or hashing on protocol headers (e.g., Receive Side Scaling). [73, 76] Commodity NICs also often support timestamping and rate control in hardware. This allows us to fulfill (R1) without violating (R4).

MoonGen does not run on arbitrary commodity hardware, we are restricted to hardware that is supported by DPDK [96] and that offers support for these features. We currently support hardware features on Intel chips of the igb, ixgbe, and i40e family. Other NICs that are supported by DPDK but not yet explicitly by MoonGen can also be used, but without hardware timestamping and rate control.

5.3.4 Software Architecture

MoonGen's core is a Lua wrapper for DPDK that provides utility functions required by a packet generator. The MoonGen API comes with functions that configure the underlying hardware features like timestamping and rate control. About 80% of the current code base is written in Lua, the remainder in C and C++. Although our current focus is on packet generation, MoonGen can also be used for arbitrary packet processing tasks.

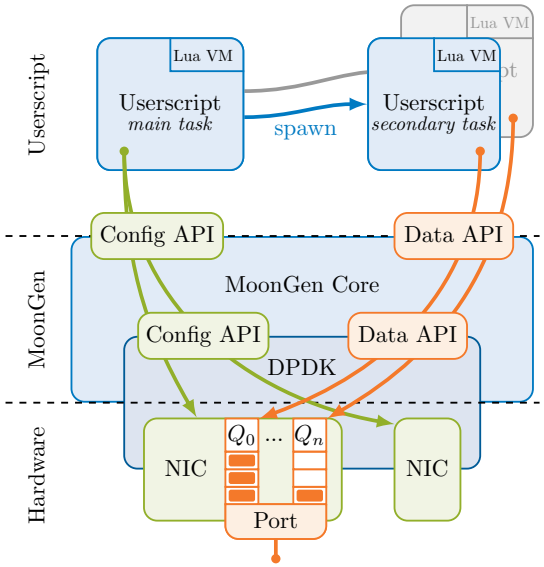


Figure 5.1: MoonGen's architecture

Figure 5.1 shows the architecture of MoonGen. It runs a user-provided script, the *userscript*, on start-up. This script contains the main loop and the packet generation logic.

The userscript will be executed in the *main task* initially by calling the *master* function provided by the script. This function must initialize the used NICs, i.e., configure the number of hardware queues, buffer sizes and filters for received traffic. It can then spawn new instances of itself running in *secondary tasks* and pass arguments to them. It usually receives a hardware queue as an argument and then transmits or receives packets via this queue. Starting a new task spawns a completely new and independent LuaJIT VM that is pinned to a CPU core. This new VM executes the same Lua script with different arguments, the MoonGen dispatcher calls the appropriate main function for the new task on startup. Tasks only share state through the underlying MoonGen library which offers inter-task communication facilities such as pipes. All functions related to packet transmission and reception in MoonGen and DPDK are lock-free to allow for multi-core scaling.

5.4 Scripting API

Our example scripts in the git repository are designed to be self-explanatory exhaustive examples for the MoonGen API [3]. The best way to understand how to use MoonGen is reading one of the example scripts in our repository [3]. We attached two example

```lua
function master(args)
    -- Configure devices.
    local txDev = device.config{port = args.txDev, rxQueues = 1, txQueues = 2}
    local rxDev = device.config{port = args.rxDev, rxQueues = 2}
    -- Wait for link on layer 1.
    device.waitForLinks()
    -- Setup hardware rate control.
    txDev:getTxQueue(0):setRate(args.bgRate)
    txDev:getTxQueue(1):setRate(args.fgRate)
    -- Start tasks.
    mg.startTask("loadTask", txDev:getTxQueue(0), 42) -- Use UDP port 42 on tx queue 0.
    mg.startTask("loadTask", txDev:getTxQueue(1), 43) -- Use UDP port 43 on tx queue 1.
    mg.startTask("counterTask", rxDev:getRxQueue(0))
    mg.waitForTasks() -- Wait until all sub-tasks finish or ctrl-c is pressed.
end
```

Listing 1: Initialization and device configuration

scripts in the appendix of this thesis. Appendix A is a configurable packet generator script that can be used as a starting point for your own scripts.

Appendix B is out quality of service example script which we use in this Section to explain and showcase features. The listings in this section show modified excerpts from this script which at its core consists of two transmission tasks to generate two types of UDP flows and measures their throughput and latencies. It can be used as a starting point for a test setup to benchmark a forwarding device or middlebox that prioritizes real-time traffic over background traffic.

The example code in this section has been modified from the example code in the repository and appendix: it has been edited for brevity. Command-line handling, error handling, and log messages are omitted The timestamping task has been removed as this example focuses on the basic packet generation and configuration API. Comments have been changed and some variables renamed, see Appendix B for the full code.

5.4.1 Initialization

Listing 1 shows the `master` function. This function is executed in the main task on startup and receives parsed command line arguments passed to MoonGen: The devices and transmission rates to use in this case. It configures one transmit device with two transmit queues and one receiving device with the default settings. The call in line 6 waits until the links on all configured devices are established. It then configures hardware rate control features on the transmission queues and starts three tasks, the first two generate traffic, the last counts the received traffic on the given device. The arguments passed to `mg.startTask` are passed to the respective functions in the new task. The `loadTask` function takes the transmission queue to operate on and a port to distinguish background from prioritized traffic.

5.4.2 Packet Generation Loop

Listing 2 shows the `loadTask` function that is started twice and does the actual packet generation. It first allocates a memory pool, a DPDK data structure in which packet buffers are allocated. The MoonGen wrapper for memory pools expects a callback function that is called to initialize each packet. This allows a script to fill all packets with default values (lines 5 to 10) before the packets are used in the transmit loop (lines 17 to 24). The transmit loop only needs to modify a single field in each transmitted packet (line 25) to generate packets from randomized IP addresses.

Line 15 initializes a packet counter that keeps track of transmission statistics and prints them in regular intervals. MoonGen offers several types of such counters with different methods to acquire statistics, e.g., by reading the NICs statistics registers. This example uses the simplest type, one that must be manually updated.

Line 17 allocates a `bufArray`, a thin wrapper around a C array containing packet buffers. This is used instead of a normal Lua array for performance reasons. It contains a number of packets in order to process packets in batches instead of passing them one-by-one to the DPDK API. Batch processing is an important technique for high-speed packet processing [53, 148].

The main loop starts in line 19 with allocating packets of a specified size from the memory pool and storing them in the packet array. It loops over the newly allocated buffers (line 22) and randomizes the source IP (line 25). Finally, checksum offloading is enabled (line 28) and the packets are transmitted (line 30).

Note that the main loop differs from a packet generator relying on a classic API. MoonGen, or any other packet generator based on a similar framework, cannot simply re-use buffers because the transmit function is asynchronous. Passing packets to the transmit function merely places pointers to them into a memory queue, which is accessed by the NIC later [96]. A buffer must not be modified after passing it to DPDK. Otherwise, the transmitted packet data may be altered if the packet was not yet fetched by the NIC.

Therefore, we have to allocate new packet buffers from the memory pool in each iteration. Pre-filling the buffers at the beginning allows us to only touch fields that change per packet in the transmit loop. Packet buffers are recycled by DPDK in the transmit function, which collects packets that were sent by the NIC earlier [96]. This does not erase the packets' contents. Our previously discussed network driver ixy works the same way, it comes with a minimal packet generation example which implements the same concept and can serve as a minimal example of the underlying concepts.

```lua
local PKT_SIZE = 60
function loadTask(queue, port)
    -- Memory pool with pre-filled packet buffers.
    local mem = memory.createMemPool(function(buf)
        buf:getUdpPacket():fill{
            pktLength = PKT_SIZE,
            ethSrc = queue, -- get MAC from device
            ethDst = "10:11:12:13:14:15",
            ipDst = "192.168.1.1",
            udpSrc = 1234,
            udpDst = port,
        }
    end)
    -- Log statistics.
    local txCtr = stats:newManualTxCounter("Port " .. port, "plain")
    -- BufArrays contain batches of packet buffers.
    local bufs = mem:bufArray()
    local baseIP = parseIPAddress("10.0.0.1")
    while mg.running() do
       -- Allocate a batch of packet buffers from the memory pool.
        bufs:alloc(PKT_SIZE)
        for _, buf in ipairs(bufs) do
            -- Randomize last byte of source IP.
            local pkt = buf:getUdpPacket()
            pkt.ip.src:set(baseIP + math.random(255) - 1)
        end
        -- Use hardware checksum offloading.
        bufs:offloadUdpChecksums()
        -- Transmit and update statistics.
        local sent = queue:send(bufs)
        txCtr:updateWithSize(sent, PKT_SIZE)
    end
    txCtr:finalize()
end
```

Listing 2: Packet transmission task with counter

5.4.3 Packet Counter

Listing 3 shows how to use MoonGen's packet reception API to measure the throughput of the different flows by counting the incoming packets.

The task receives packets from the provided queue in the `bufArray` `bufs` in line 5. It then extracts the UDP destination port from the packet (line 9) and uses counters to track statistics per port. The final statistics are printed by calling the counters' finalize methods in line 24. Printed statistics include the average packet and byte rates as well as their standard deviations.

The format to print in is specified in the counter constructor in line 13. All example scripts use the `plain` formatter, the default value is `CSV` for easy post-processing. The output can also be diverted to a file. Details are in the LuaDoc documentation of `stats.lua`.

This script can be used for another similar test setup by adapting the code to the test setup by changing hardcoded constants like the used addresses and ports. The full

```lua
function counterTask(queue)
    local bufs = memory.bufArray()
    local counters = {}
    while mg.running() do
        local rx = queue:recv(bufs)
        for i = 1, rx do -- For each received packet.
            local buf = bufs[i]
            -- Extract UDP destination port.
            local port = buf:getUdpPacket().udp:getDstPort()
            -- One counter for each port, created dynamically
            local ctr = counters[port]
            if not ctr then
                ctr = stats:newPktRxCounter(port, "plain")
                counters[port] = ctr
            end
            -- Track the packet.
            ctr:countPacket(buf)
        end
        -- Free packet memory.
        bufs:freeAll()
    end
    -- Print final statistics.
    for _, ctr in pairs(counters) do
        ctr:finalize()
    end
end
```

Listing 3: Packet counter task

script in the repository [3] includes a separate timestamping task to acquire and print latency statistics for the two flows.

5.5 Performance

Writing the whole generation logic in a scripting language raises concerns about the performance, but we are able to achieve our performance targets from Section 2.2 of 14.88 Mpps on a single core. One important feature of LuaJIT is that it allows for easy integration with existing C libraries and structs: it can directly operate on C structs and arrays without incurring overhead for bound checks or validating pointers [123]. Thus, crafting packets is very efficient in MoonGen.

The obvious disadvantage is that unchecked memory accesses can lead to memory corruption, a problem that is usually absent from scripting languages. However, most critical low-level parts like the implementation of the NIC driver are handled by DPDK. The MoonGen core then wraps potentially unsafe parts for the userscript if possible. There are only two operations in a typical userscript that can lead to memory corruption: writing beyond packet buffer boundaries and trying to operate on buffers that are null pointers. This is an intentional design decision to aid the performance: we are not interested in the safety properties of high-level languages here but in their flexibility and ease of use.

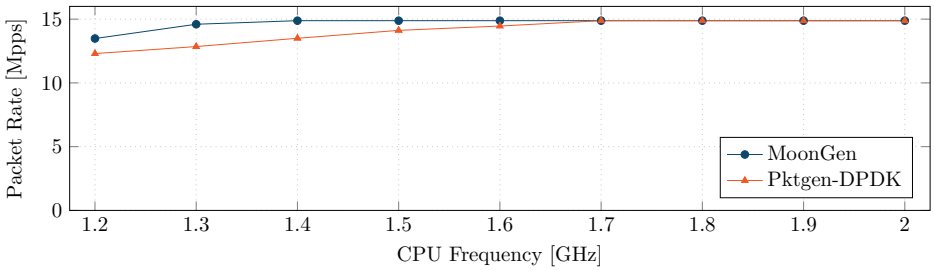


Figure 5.2: CPU frequency vs. generated packets per second

5.5.1 Test Methodology

DPDK works—like our ixy driver—with busy-wait loops and always puts 100% on all configured CPU cores. We therefore use a test setup that enforces a CPU bottleneck similar to our driver evaluation: we adjust the CPU clock frequency until the evaluated packet generators fail to achieve line rate to quantify performance in CPU cycles per packet.

The tests in this section were executed on an Intel Xeon E5-2620 v3 CPU with a frequency of 2.4 GHz that can be clocked down to 1.2 GHz in 100 MHz steps (Server 1 from Section 2.4). To ensure consistent and reproducible measurement results, we disabled Hyper-Threading, which may influence results if the load of two virtual cores is scheduled to the same physical core. TurboBoost and SpeedStep were also disabled because they adjust the clock speed according to the current CPU load and interfere with our manual adjustment of the frequency.

5.5.2 Comparison with Pktgen-DPDK

Our scripting approach can even increase the performance compared to a static packet generator slightly. We show this by comparing MoonGen to `pktgen-dpdk` [165], a packet generator for DPDK written in C. Both MoonGen and the version of `pktgen-dpdk` used here were build on the same version of DPDK.

We configured both packet generators to craft minimum-sized UDP packets with 256 varying source IP addresses on a single CPU core. We then gradually increased the CPU's frequency until the software achieved line rate. `pktgen-dpdk` required 1.7 GHz to hit the 10 Gbit/s line rate of 14.88 Mpps, MoonGen only 1.5 GHz. `pktgen-dpdk` achieved 14.12 Mpps at 1.5 GHz. This means MoonGen is more efficient in this specific scenario.

This increased performance is an inherent advantage of MoonGen's architecture `pktgen-dpdk` needs a complex main loop that covers all possible configurations even though we

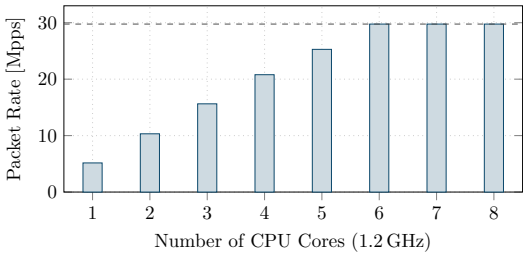


Figure 5.3: Multi-core scaling under high load

are only interested in changing IP addresses in this test scenario. MoonGen, on the other hand, can use a script that consists of a tight inner loop that exclusively executes the required tasks: allocating pre-filled packet buffers, modifying the IP address, and sending the packets with checksum offloading. You only pay for the features you actually use with MoonGen.

5.5.3 Multi-core Scaling

The achieved performance depends on the script; the previous example was a light workload for the comparison to `pktgen-dpdk`, which is limited to such simple patterns. Therefore, we test a more involved script to stress MoonGen to show the scaling with multiple cores sending to the same NIC via multiple transmission queues.

Figure 5.3 shows the performance under heavy load and the scaling with the number of CPU cores. MoonGen was configured to generate minimum-sized packets with random payload as well as random source and destination addresses and ports. The code generates 8 random numbers per packet to achieve this. Each core generated and sent packets on two different 10 Gbit/s interfaces simultaneously. Linear scaling can be observed up to the line rate limit (dashed line). The trade-off is that we use the NIC's multi-queue hardware feature (Section 2.1) is required, meaning we lose control over the exact order in which packets are sent.

The code was written in idiomatic Lua without specific optimizations for this use case: LuaJIT's standard random number generator, a Tausworthe generator with a period of 2^{223}, was used [123]. Since a high quality random number generator is not required here, a simple linear congruential generator would be faster. The code also generates a random number per header field instead of combining multiple fields (e.g., source and destination port can be randomized by a single 32-bit random number).

Despite the lack of optimizations, the code was initially found to be too fast for meaningful scalability measurements (10.3 Mpps on a single core). We therefore reduced the CPU's clock speed to 1.2 GHz and increased the number of NICs to 2 for this test.

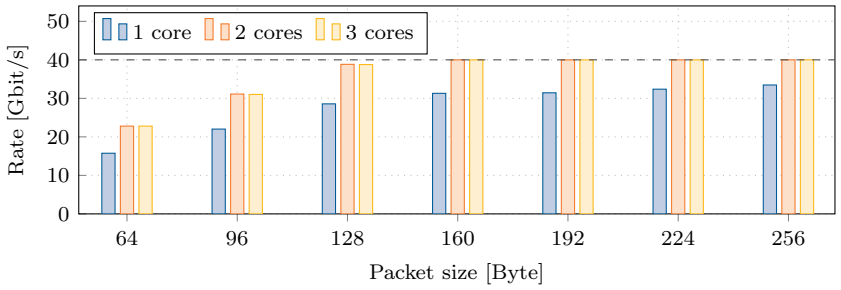


Figure 5.4: Throughput with an XL710 40 Gbit/s NIC

This test shows that sending to a single NIC port via multiple queues scales linearly, an important assumption made for our architecture (cf. Section 5.3.3).

5.5.4 Scaling to 40 Gigabit Ethernet

40 Gbit/s NICs like the dual port Intel XL710 [77] are nowadays common in many servers. However, these NICs come with bandwidth limitations that do not exist on the 10 Gbit/s NICs discussed previously: they cannot saturate a link with minimum-sized packets [137] and they cannot saturate both ports simultaneously regardless of the packet size [77]. This may limit their use in some scenarios where a large number of small packets is required, e.g., stress-testing a router.

Figure 5.4 shows the achieved throughput with various packet sizes and number of 2.4 GHz CPU cores used to generate the traffic. Packet sizes of 128 bytes or less cannot be generated in line rate. Using more than two CPU cores does not improve the speed, so this is a hardware bottleneck as described by Intel [137].

The second bandwidth restriction of this NIC is the aggregate bandwidth of the two ports. One obvious restriction is the 63 Gbit/s bandwidth of the PCIe 3.0 x8 link that connects the NIC to the CPU. However, the main bottleneck is the media access control layer in the XL710 chip: it is limited to a maximum aggregate bandwidth of 40 Gbit/s (cf. Section 3.2.1 of the XL710 datasheet [77]). We could achieve a maximum bandwidth of 50 Gbit/s with large packets on both ports simultaneously and a maximum packet rate of 42 Mpps (28 Gbit/s with 64 byte frames).

5.5.5 Scaling to 100 Gigabit Ethernet

We equipped one of our test servers with six dual-port 10 Gbit/s Intel X540-T2 NICs to investigate the performance at high rates. Figure 5.5 shows the achieved packet rate when generating UDP packets from varying IP addresses. We used two Intel Xeon E5-

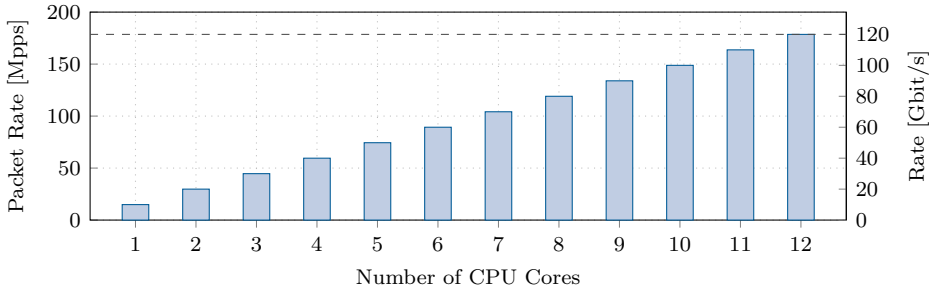


Figure 5.5: Multi-core scaling (multiple 10 Gbit/s NICs)

2640 v2 CPUs with a nominal clock rate of 2 GHz for this test, but the clock rate can even be reduced to 1.5 GHz for this packet generation task (cf. Section 5.5.2).

Note that sending to multiple NICs simultaneously is architecturally the same as sending to multiple queues on a single NIC as different queues on a single NIC are independent from each other (cf. Section 5.5.3) in an ideal well-behaved NIC like the current generation of 10 Gbit/s NICs. However, the currently available 100 Gbit/s NICs have similar hardware restrictions as the 40 Gbit/s NIC discussed in Section 5.5.4. For example, the Mellanox ConnectX-4 cannot handle transmitting more than 67 Mpps (only 45 Gbit/s with minimum sized packets) even under ideal conditions with multiple queues [59].

5.5.6 Per-Packet Costs

MoonGen's dynamic approach to packet generation in userscripts does not allow for a performance analysis in a general configuration as there is no typical scenario. Nevertheless, the cost of sending a packet can be decomposed into three main components: packet IO, memory accesses, and packet modification logic. We devised a synthetic benchmark that measures the average number of CPU cycles required for various operations that are commonly found in packet generator scripts. These measurements can be used to estimate the hardware requirements of arbitrary packet generator scripts.

The full benchmarking script can be found in commit `96818ad9` in our git repository [38]. It works by chaining together different packet modification operations and then transmitting packets over a period of 8 seconds, counting the packets transmitted. Measuring the packet rate every second during a run shows a deviation of less than 0.1%. Yet, when repeating this experiment we observe a significant change in throughput depending on the complexity of the modification operations. Hence, we repeat all measurements of packet costs 10 times here, the uncertainties given in this section are the standard deviations.

Operation	Cycles/Pkt
Packet transmission	76.0 ± 0.8
Packet modification (constants within the first 64 bytes)	9.1 ± 1.2
Packet modification (constants within the first 128 bytes)	15.0 ± 1.3
IP checksum offloading	15.2 ± 1.2
UDP checksum offloading	33.1 ± 3.5
TCP checksum offloading	34.0 ± 3.3

Table 5.1: Per-packet costs of basic operations (2.4 GHz CPU, performance budget of 161 cycles)

Basic Operations

Table 5.1 shows the average per-packet costs of basic operations for IO and memory accesses. The baseline for packet IO consists of allocating a batch of packets and sending them without touching their contents in the main loop. This shows that there is a considerable per-packet cost for the IO operation caused by the underlying DPDK framework, see Section 3.5.6 for a breakdown.

Modification operations write constants into the packets, forcing the CPU to load them into the layer 1 cache. Additional accesses within the same cache line (64 bytes) add no measurable additional cost. Accessing another cache line in a larger packet is noticeable.

Offloading checksums is not free (but still cheaper than calculating them in software) because the driver needs to set several bitfields in the DMA descriptor. For UDP and TCP offloading, MoonGen also needs to calculate the IP pseudo header checksum as this is not supported by the X540 NIC used here [76].

Randomizing Packets

Sending varying packets is important to generate different flows. There are two ways to achieve this: one can either generate a random number per packet or use a counter with wrapping arithmetic that is incremented for each packet. The resulting value is then written into a packet header at a fixed offset. Table 5.2 shows the cost for the two approaches, the baseline is the cost of writing a constant to a packet and sending it (85.1 cycles/pkt).

There is a fixed cost for calculating the values while the marginal cost is relatively low: 17 cycles/packet per random field and 1 cycle/packet for wrapping counters. These results show that wrapping counters instead of actual random number generation should be preferred if possible for the desired traffic scenario.

We use the default random number generator in LuaJIT, Tausworthe generator with a period of 2^{223} [123]. Since this randomization happens in user-defined code one is free

Memory locations	Cycles/Pkt (Rand)	Cycles/Pkt (Counter)
1	32.3 ± 0.5	27.1 ± 1.4
2	39.8 ± 1.0	33.1 ± 1.3
4	66.0 ± 0.9	38.1 ± 2.0
8	133.5 ± 0.7	41.7 ± 1.2

Table 5.2: Per-packet costs of modifications on 4 byte fields at fixed offsets (2.4 GHz CPU, performance budget of 161 cycles

to use a simpler custom random number generation if the quality of randomness is not a concern. Further, MoonGen supports playing back pcap files, so precomputed packet streams can be sent out.

Cost Estimation Example

We can use these values to predict the performance of the scripts used for the performance evaluation in Section 5.5.3. The example generated 8 random numbers for fields with a userscript that is completely different from the benchmarking script: it writes the values into the appropriate header fields and the payloads, the benchmarking script just fills the raw packet from the start. The script also combines offloading and modification; the benchmark tests them in separate test runs.

The expected cost consists of: packet IO, packet modification, random number generation, and IP checksum offloading, i.e., 229.2 ± 3.9 cycles/pkt. This translates to a predicted throughput of 10.47 ± 0.18 Mpps on a single 2.4 GHz CPU core. The measured throughput of 10.3 Mpps is within that range. This shows that our synthetic benchmark can be used to estimate hardware requirements.

It should be noted that this model has limitations: it assumes that multiple operations are independent from each other. This might be true logically, but on a lower layer several effects need to be taken into account including but not limited to optimizations applied by out-of-order CPUs and effects of the cache.

5.5.7 Effects of Packet Sizes

All tests performed in the previous sections use minimum-sized packets. The reason for this choice is that the per-packet costs dominate over costs incurred by large packets. Allocating and sending larger packets without modifications add no additional cost in MoonGen on 1 and 10 Gbit/s NICs. Only modifying the content on a per-packet basis adds a performance penalty, which is comparatively low compared to the fixed cost of sending a packet. Using larger packets also means that fewer packets have to be sent at

line rate, so the overall fixed costs for packet IO are reduced: minimum-sized packets are usually the worst-case.

Nevertheless, there are certain packet sizes that are of interest: those that are just slightly larger than a single cache line. We benchmarked all packet sizes between 64 and 128 bytes and found no difference in the CPU cycles required for sending a packet. Larger packets hit the line rate even with the lowest supported CPU clock frequency of 1.2 GHz. Since MoonGen also features packet reception, we also tried to receive packets with these sizes and found no measurable impact of the packet size.[1]

Rizzo notes that such packet sizes have a measurable impact on packet reception, but not transmission, in his evaluation of netmap [148]. He attributes this to hardware bottlenecks as it was independent from the CPU speed. We could not reproduce this with MoonGen. The likely explanation is that we are using newer server hardware (CPU launched in 2014), while the evaluation of netmap was done on an older system with a CPU launched in 2009 [148].

5.6 Hardware Timestamping

Another important performance characteristic beside the throughput is the latency of a system. Modern NICs offer hardware support for the IEEE 1588 Precision Time Protocol (PTP) for clock synchronization across networks. PTP can be used either directly on top of Ethernet as a layer 3 protocol with EtherType 0x88F7 or as an application-layer protocol on top of UDP [68].

We examined the PTP capabilities of the Intel 82580 1 Gbit/s NICs, the Intel 82599, X540, and X550 10 Gbit/s chips as well as the XL710 40 Gbit/s NICs. They support timestamping of PTP Ethernet and UDP packets, the UDP port is configurable on the 10 and 40 Gbit/s NICs. They can be configured to timestamp only certain types of PTP packets, identified by the first byte of their payload. The second byte must be set to the PTP version number. All other PTP fields in the packet are not required to enable timestamps and may contain arbitrary values. [72, 73, 76, 77] This allows us to measure latencies of almost any type of packet.

Most Intel NICs, including the 10 and 40 Gbit/s chips used here, save the timestamps for received and transmitted packets in a register on the NIC. This register must be read back before a new packet can be timestamped [73, 76], limiting the throughput of

[1] Note that this is not true for XL710 40 Gbit/s NICs which can run into hardware bottlenecks with some packet sizes (Section5.5.4).

timestamped packets. The XL710 offers 4 registers that can be used in a queue, offering a slightly higher throughput but still one additional PCIe round trip for every 4 packets. However, none of these approaches is fast enough to handle timestamping all packets – PCIe round trips are prohibitively expensive, see the discussion in Chapter 3 where the main goal was avoiding PCIe accesses by batching packets.

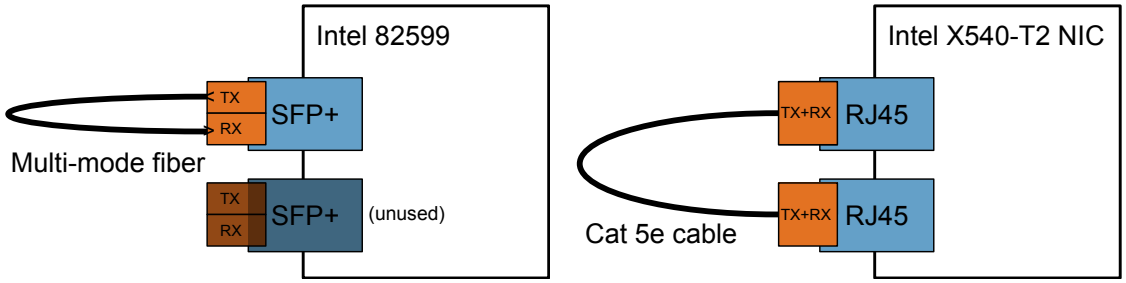
Timestamping all packets on transmission is not feasible on any evaluated commodity NIC as none supports the insertion of the timestamp into the packet. But there are NICs that support appending timestamps of all received packets to the DMA buffers. MoonGen supports this by providing a small driver (as the DPDK driver does not support this) to enable this feature for the Intel 82580, X550, and Intel X552 NICs. This has proven especially useful on the X552 NICs which are available with SFP+ ports and hence can be used to timestamp all packets on passive tap devices. This setup effectively removes the requirement to acquire timestamps on transmission (infeasible on commodity hardware) and turns it into timestamping on reception at two observation points, which is supported on selected NICs. See Section 2.4.1 for a description of this test setup which allows us to measure the timestamps of all packets, this was used for the latency measurements presented in Section 4.7.

5.6.1 Precision and Accuracy

We use the Intel 82599 and X540 NICs as examples for the evaluation of timestamping as they represent two different physical layers: the former NIC has a SFP+ slot that can be used with fiber optics, the latter a normal RJ45 port used with Cat 5e cabling.

Timestamping mechanisms of the Intel 82599 and Intel X540 10 Gbit/s chips operate at 156.25 MHz when running at 10 Gbit/s speeds [73, 76]. This frequency is reduced to 15.625 MHz when a 1 Gbit/s link is used, resulting in a granularity of 6.4 ns for 10 Gbit/s and 64 ns for 1 Gbit/s. The datasheet of the Intel 82580 Gbit/s [72] controller lacks information about the clock frequency, testing shows the acquired timestamps are always a multiple of 64 ns.

All of these NICs timestamp packets late in the transmit path and early in the receive path so as to be as accurate as possible [72, 73, 76]. The timestamp is taken when the last bit of the start-of-frame delimiter of an Ethernet packet is transmitted/received. For the 82599 and X540 evaluated here, the exact location within the pipeline where this happens is not specified. The datasheet merely describes it as “as close as possible to the PHY” [73, 76].



(a) Intel 82599 with a loopback cable

(b) Intel X540 connected to itself

Figure 5.6: Loopback configurations to measure the latency incurred by a cable

NIC	t_{2m} [ns]	$t_{8.5m}$ [ns]	t_{10m} [ns]	t_{20m} [ns]	t_{50m} [ns]	k [ns]	v_p
82599 (10GBASE-SR)	320	352	-	403.2	-	310.7 ± 3.9	$0.721c^{+0.117c}_{-0.087c}$
X540 (10GBASE-T)	2156.8	-	2195.2	-	2387.2	2147.2 ± 4.8	$0.69c^{+0.236c}_{-0.037}$

Table 5.3: Timestamping accuracy measurements (±6.4 ns precision)

We can evaluate the precision by using a simple cable as a "system" under test. The latency t of a cable can be described with the following formula.

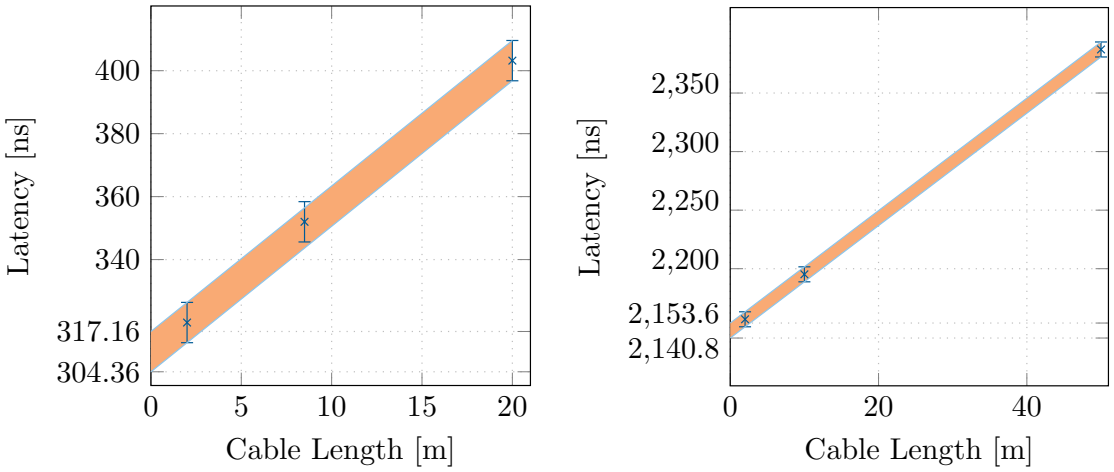
$$t = k + l/v_p$$

Where k is some constant systematic error introduced by the NIC (e.g., (de-)modulation time), l is the length of the cable, and v_p is the propagation speed of the signal in the selected medium. For the length of the cable we rely on the vendor's specification.

Timestamping on the 82599 with 10GBASE-SR

Since 10GBASE-SR uses dedicated fibers for each direction we can connect a port to itself as shown in Figure 5.6a. We measure the latency of OM3 multimode fiber cables with the length 2 m, 8.5 m and 20 m, taking 500 000 samples for each cable. For the 2 m and 20 m cabling we measure latencies of 320 ns and 403.2 ns respectively with a standard deviation of 0. The 8.5 m cable measures a time of 345.6 ns in 50.2% of the measurements and 358.4 ns in the other 49.8%, i.e., a mean of 352 ns with a standard deviation of 6.4 ns. Table 5.3 summarizes these results.

Figure 5.7a plots the measurement results, the error bars are the ±6.4 ns incurred by the timestamping granularity for the 2 m and 20 m case, and the same 6.4 ns of measured standard deviation for the 8.5 m cable. The shaded orange area connects the upper and lower bounds for the shortest and longest cable measured. The intermediate-length



(a) Intel 82599 with a loopback cable

(b) Intel X540 connected to itself

Figure 5.7: Latencies of different cables lengths, 500 000 measurements per data point.

cable falls within that area, indicating the expected linear relationship between cable length and measured time.

Calculating the slope of the line connecting the measurements of the shortest and longest cables yields a value of 4.62 $\frac{m}{ns}$ or 72.1% of the speed of light c as v_p. The slope from the lower end of the shorter cable to the upper end of the longer cable and vice versa yields error bars of this quantity of 63.4% to 83.8% of the speed of light. Extending the line until it intersects the y-axis gives us the constant offset k, or systematic error caused by the NIC, as 310.76±6.4 ns.

We can compare the result for the propagation speed with the theoretical value. The fibers we use have a specified refractive index of 1.482 at the wavelength of 850 nm used here. This corresponds to a propagation speed of 67.5% the speed of light. In addition, some additional delay is caused by internal reflection of the signal inside the wire. However, the internal refraction happens at a shallow angle: critical angle for total internal reflection is specified as 81.5° on the cable used here. This corresponds to an extra path length of up to 1.11%, i.e., an effective propagation speed of 66.75% the speed of light, matching the common textbook value of 66.7% [131]. This puts the theoretical value within the measurement uncertainty of our value.

Timestamping on the X550 with 10GBASE-T

Doing the same for the X550 NIC requires us to use two different ports which does not affect this experiment as there is no clock drift between two ports on the same

NIC (Section 5.6.3). We use Cat 5e cables of length 2 m, 10 m, and 50 m. Table 5.3 summarizes these results.

Measurement results on the 10GBASE-T interface are noiser than the ones from the previous section. Repeating the measurements 500 000 times as before shows a difference between 64 ns between the lowest and highest measured latency with the same cable. A likely reason for this is that 10GBASE-T uses a complex block code on layer 1 [67] which introduces this variance. Yet, 99.5% of the measured values are within ±6.4 ns of the mean, the standard deviation is less than the theoretical error of the 6.4 ns incurred from the granularity, Figure 5.7b therefore shows error bars of ±6.4 ns.

Despite this lack of precision, the mean measured values for the 3 cables are *exactly* on a straight line with a slope of 4.8 ns/m (69.5% the speed of light with a possible range of 65.8% to 89.4% when taking the error into account). This straight line indicates a low random error component of the accuracy. The theoretical value for propagation speed of signals in Cat 5e cables cannot be calculated easily. Multiple values can be found in the literature. A value of 77% is given in commonly used networking textbooks [131], however, it does not specify the exact type of cable used here. Other sources indicate that this value might refer to thick coax cables as used in 10BASE5 Ethernet [86]. The same source gives a propagation speed of 59% for unshielded twisted pair, similar to the shielded Cat 5e wire used here. Another source claims that the "typical" propagation speed in Cat 5e is 67% of the speed of light, but also gives a worst case of 59% [47]. Our measured value is close to this typical propagation speed, however, without a good source of truth we cannot make any strong claims based on this result, only that our measured value is plausible.

The constant offset or systematic error component of the accuracy is 2 147.2±6.4 ns. This large value is is expected, others have reported an "approximate" constant latency penalty of 2.5 µs for 10GBASE-T [166].

Conclusion

We claim a precision of only ±12.8 ns for the 82599 NIC because of the measurement result with the 8.5 m cable. The reason for this can be found in the datasheet of the 82599 NIC explaining that while the timestamping logic itself operates at the full 156.25 MHz frequency, the counter that is saved when the timestamp is taken can only be incremented every other clock cycle [73]. While the X540 NIC does not suffer from this problem, we can only claim a precision of ±32 ns because of the observed outliers in 0.05% of the measurements.

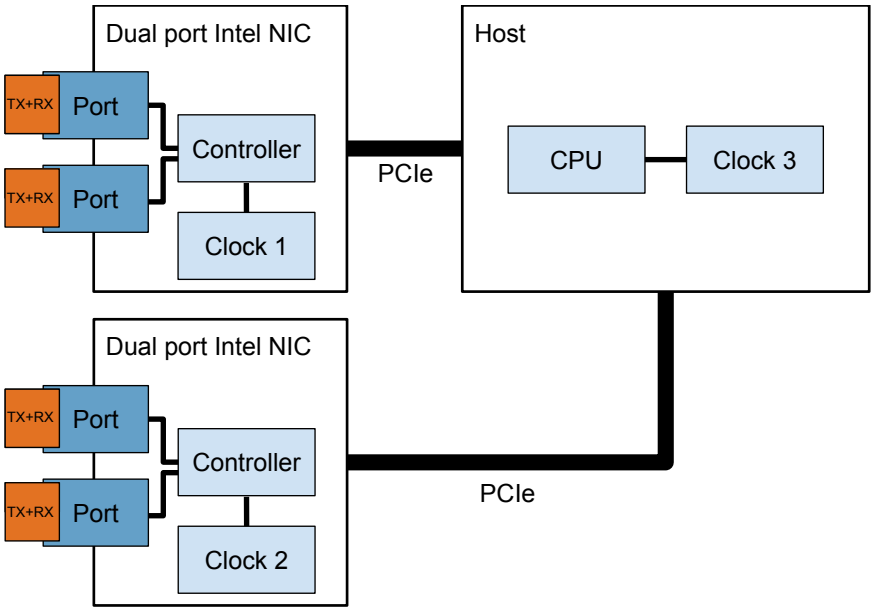


Figure 5.8: Different clocks in a server running MoonGen. Timestamps are taken with Clock 1 and Clock 2.

Based on these results we recommend to use a fiber connection on an 82599 NIC for latency measurements. It is not only more precise, but also more accurate: Its systematic error of 317 ns is 7 times lower than the error of the X540 NIC.

5.6.2 Clock Synchronization

A typical server used for measuring latency with MoonGen features many independent clocks, see Figure 5.8. Hardware timestamping must rely on the clocks found on the NICs to be close to the actual reception and transmission of packets. MoonGen therefore needs to be able to synchronize the clocks between different NICs. This is even necessary between two ports of a dual-port NIC, which are completely independent from the user's point of view because the NIC internally uses different epochs for the two devices exposed on the PCIe bus.

MoonGen synchronizes the clocks of two ports by reading the current time from both clocks and calculating the difference. The clocks are then read again in the opposite order. The resulting differences are the same if and only if the clocks are currently synchronous (assuming that the time required for the PCIe access is constant). We observed randomly distributed outliers in about 5% of the reads. We therefore repeat the measurement 7 times to have a probability of $> 99.999\%$ of at least 3 correct measurements. The median of the measured differences is then used to adjust one of the clocks to synchronize them. This adjustment must be done with an atomic read-

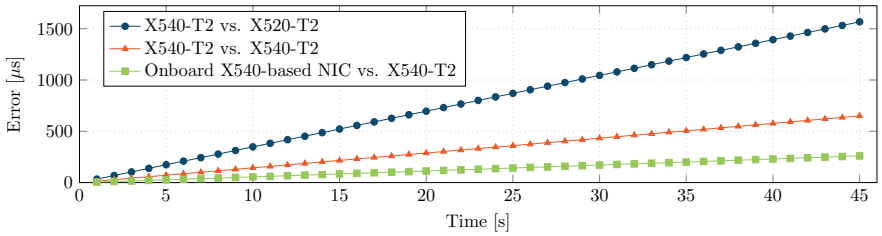


Figure 5.9: Clock drift between different NICs

modify-write operation. The NICs support this as it is also required to implement PTP.

Tests show that this technique synchronizes the clocks with an error of ±1 cycle. Therefore, the best possible accuracy for tests involving multiple network interfaces is 19.2 ns for the 10 Gbit/s chips.

5.6.3 Clock Drift

Using two different clocks also entails the risk of clock drifts. Drift on X540-based NICs depends on the physical wiring as the timestamping clock is synchronized to the physical layer. Two ports on different X540-based NICs that are directly connected do not exhibit any clock drift while the link is established. We did not observe clock drifts between the two ports of 82599-based dual port NICs, indicating that they always use the same clock source for both ports (see Figure 5.8).

Figure 5.9 shows clock drifts between several different NICs, all NICs were left unconnected during the test. The worst-case observed drift was 35 µs per second, i.e., 35 ppm between a X540 NIC integrated on the server's main board and one on a PCIe card. Note that the clock drift was stable in this short 45 second test which is in the range of test lengths typically used for MoonGen. Long-term stability may be affected by temperature differences.

MoonGen handles clock drift by resynchronizing the clocks (Section 5.6.2) before a timestamped packet is sent, so this drift translates to a maximum relative error of only 0.0035% that does not accumulate over time. This is not significant for latency measurements: for a typical measurement in the 100 µs range this corresponds to an error of only 3.5 ns due to this effect. Using a dual-port NIC avoids this problem in the first place as both clocks share a physical clock, so there is no drift, see Figure 5.9. In practice, we recommend this dual-port NIC setup for latency measurements as other NICs might have worse clock drift than the ones we evaluated here.

5.6.4 Limitations

Our approach for latency measurements comes with limitations. The latency measurements are restricted to Ethernet frames with the PTP EtherType and UDP packets. MoonGen cannot measure latencies of other protocols. Note that we do not actually use any features of the PTP protocol. We just craft packets that look like PTP packets to the NIC to trigger the timstamping hardware, the UDP port it triggers on is configurable.

The naïve handling of clock drift by resynchronizing the clocks for each packet allows for only a single timestamped packet in flight, limiting the throughput to $1\,Pkt/RTT$. MoonGen scripts therefore use two transmission queues, one that sends timestamped packets and one that sends regular packets. The regular packets can be crafted such that the system under test cannot distinguish them from the timestamped packets, e.g., by setting the PTP type in the payload to a value that is not timestamped by the NIC. So MoonGen effectively samples random packets in the data stream and timestamps them. Sampling strictly limited by RTT would introduce bias as the sampling rate would increase during periods of low latency and decrease during high latency. MoonGen avoids this by only timestamping 1000 packets every second, this value is adjustable if latencies above 1 ms are expected. Note that the benchmarking standard RFC 2544 calls for only one timestamped packet in a 120 second interval [25].

The investigated NICs refuse to timestamp UDP PTP packets that are smaller than the minimum PTP packet size of 80 bytes. Larger packets are timestamped properly. This restriction does not apply to packets with the PTP EtherType as the minimum PTP packet size is below 64 bytes in this configuration.

Based on the discussed measurement results and despite these limitations, we argue that special-purpose hardware is not necessary to conduct high-precision and accurate latency measurements for many scenarios. The very low cost of this setup is especially attractive: compatible NICs are often found onboard in server-grade mainboards, compatible PCIe add-on cards are available for around 150 euros (2021 prices).

5.7 Rate Control

An important feature of a packet generator is controlling the packet rate and generating specific timing patterns to simulate real-world scenarios. MoonGen utilizes hardware rate control features of Intel NICs to generate constant bit rate and bursty traffic. We also implement a novel software-based rate control for realistic traffic patterns, e.g., based on a Poisson process.

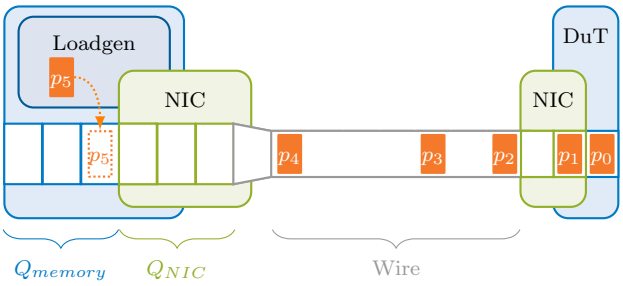


Figure 5.10: Software-based rate control (figure from [42])

5.7.1 Software Rate Control in Existing Packet Generators

Trying to control the timing between packets in software is known to be error-prone [23, 29]. The main problem with software-based rate control is that the software needs to push individual packets to the NIC and then has to wait for the NIC to transmit it before pushing the next packet.

However, modern NICs do not work that way: they rely on an asynchronous push-pull model and not on a pure push model. Chapter 3 Section 3.4 explains the packet transmission process in detail, to recap: The software writes the packets into a queue that resides in the main memory and informs the NIC that new packets are available. It is up to the NIC to fetch the packets asynchronously via DMA and store them in the internal transmit queue on the NIC before transmitting them.

Figure 5.10 visualizes this packet flow. Only a single packet at a time is allowed in the queues (Q_{memory} & Q_{NIC}) to generate packets that are not back-to-back on the wire.

This hardware architecture causes two problems: the exact timing when a packet is retrieved from memory cannot be controlled by the software and queues cannot be used (unless bursts are desired). The former results in a low precision, as the exact time when the packet is transferred cannot be determined. The latter impacts the performance at high packet rates as high-speed packet processing relies on batch processing (cf. Chapter 3 Section 3.4).

5.7.2 Hardware Rate Control

Intel 10 Gbit/s NICs feature hardware rate control: all transmit queues can be configured to a specified rate. The NIC then generates constant bit-rate (CBR) traffic. This solves the two problems identified in the previous section. The software can keep all available queues completely filled and the generated timing is up to the NIC. Figure 5.11 shows this architecture. The disadvantage is that this approach is limited to CBR traffic

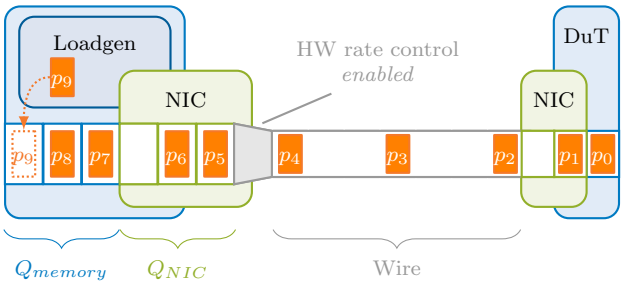


Figure 5.11: Hardware-based rate control (figure from [42])

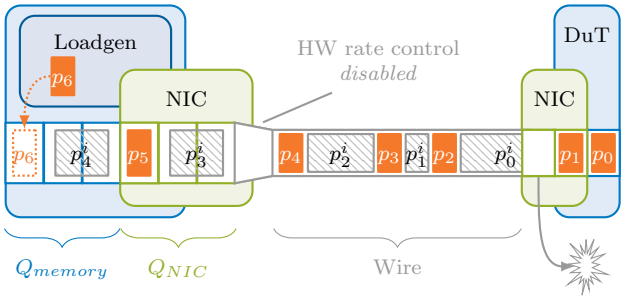


Figure 5.12: Precise generation of arbitrary traffic patterns in MoonGen (figure from [42])

and bursty traffic (by changing the rate parameter periodically) and relies on black-box hardware working as advertised (which it does not, see Section 6.6.4).

5.7.3 Controlling Inter-Packet Gaps in Software

To overcome this restriction to constant bit rate or bursty traffic, MoonGen implements a novel mechanism for software-based rate control. This allows MoonGen to create arbitrary traffic patterns.

All existing software packet generators try to delay sending packets by not sending packets for a specified time, leading to the previously mentioned problems. MoonGen fills the gaps between packets with invalid packets instead. Varying the length of the invalid packet precisely determines the time between any two packets and subsequently allows the creation of arbitrary complex traffic patterns. With this technique, we can still make use of the NIC's queues and do not have to rely on any timing related to DMA accesses by the NIC.

This approach requires support by the system under test: it needs to detect and ignore invalid packets in hardware without affecting the packet processing logic. MoonGen uses packets with an incorrect CRC checksum and, if necessary, an illegal length for short gaps. This behavior is configurable as it might pose a problem when testing hardware devices such as switches or routers.

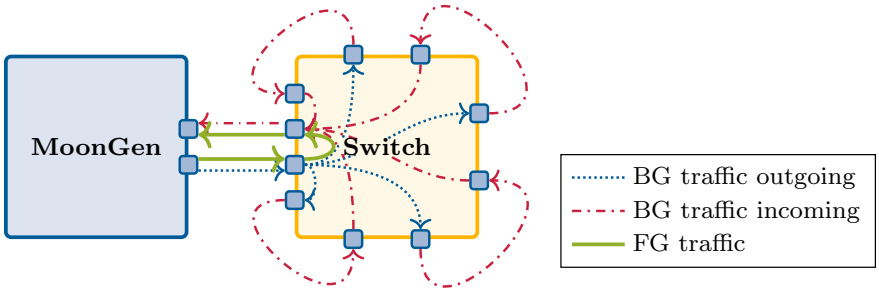


Figure 5.13: Test setup for testing a switch with MoonGen by using traffic amplification inside the switch with OpenFlow (adapted from [39])

All investigated NICs in our testbed drop such packets early in the receive flow: they are dropped even before they are assigned to a receive queue, the NIC only increments an error counter [72, 73, 76, 77]. Subsequently, the packet processing logic is not affected by this software rate control mechanism.

Figure 5.12 illustrates this concept. Shaded packets p_j^i are sent with an incorrect CRC checksum, all other packets p_k with a correct one. Note that the wire and all transmission queues are completely filled, i.e., the generated rate has to be the line rate.

In theory, arbitrary inter-packet gaps should be possible. The NICs we tested refused to send out frames with a wire-length (including Ethernet preamble, start-of-frame delimiter, and inter-frame gap) of less than 33 bytes, so gaps between 1 and 32 bytes (0.8 ns to 25.6 ns) cannot be generated. Generating small frames also puts the NIC under an unusually high load for which it was not designed. We found that the maximum achievable packet rate with short frames is 15.6 Mpps on Intel X540 and 82599 chips, only 5% above the line rate for packets with the regular minimal size. MoonGen therefore enforces a minimum wire-length of 76 bytes (8 bytes less than the regular minimum) by default for invalid packets. As a result, gaps between 0.8 ns and 60.8 ns cannot be represented.

5.8 Example: Measuring Forwarding Latency of an OpenFlow Switch

MoonGen's usefulness and precision can be demonstrated with an example measurement. We measure the latency of an Edge-Core Networks AS5712-54X 10 Gbit/s OpenFlow switch with a Broadcom BCM56854 Trident II switch ASIC [9]. This measurement is taken from our publication about traffic amplification with OpenFlow to scale MoonGen beyond 100 Gbit/s [39]. The thesis of this publication is that we can use OpenFlow

flooding rules to amplify selected flows of traffic beyond what MoonGen can generate in software.

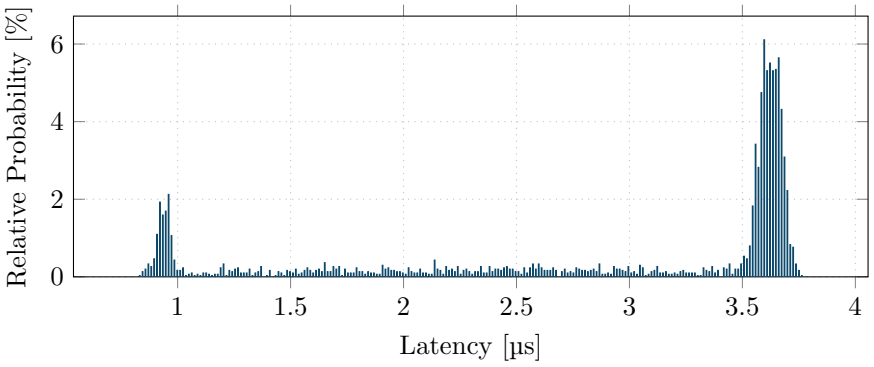
For one experiment in this publication we wired the aforementioned 48-port OpenFlow switch with itself by connecting port 1 with port 2, port 3 with port 4, etc. Ports 47 and 48 were directly connected to a server running MoonGen. All connects use short (50cm to 1m) direct attach copper SFP+ cables. We then generate two UDP flows with different UDP ports (see example script in 5.4), one flow carries background (BG) traffic, the other foreground traffic (FG). On the switch we install the following OpenFlow rules:

1. FG traffic on port 47 (from MoonGen) is sent to port 48 (to MoonGen)
2. BG traffic on port 47 (from MoonGen) is flooded to all other ports except 48
3. Incoming traffic on ports 1-46 (amplified BG traffic) is forwarded to port 48 (to MoonGen)

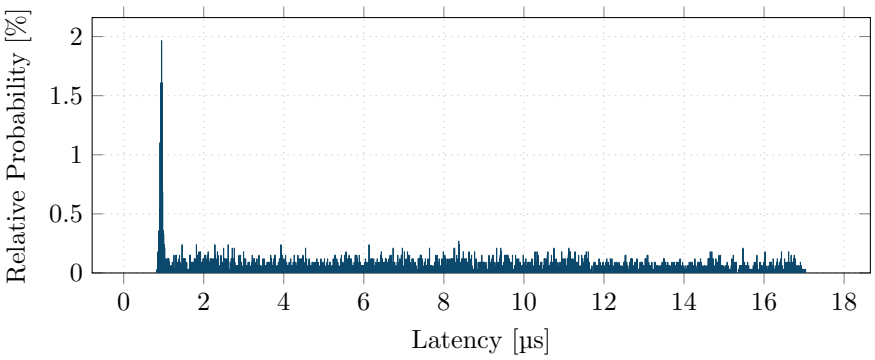
These rules together with the external wiring (ports 1-46 connected to each other) cause the switch to effectively flood itself, amplifying the background traffic 46-fold. Figure 5.13 shows this setup with a reduced number of ports and the bi-directional nature of the flooding omitted for clarity.

For this experiment we send 1.5 Mpps (1 Gbit/s) of foreground traffic and 0.26 Mpps of background traffic, which gets amplified to 11.9 Mpps (8 Gbit/s). The volume of the traffic here is less important than the fact that the incoming amplified background traffic comes from 46 other ports, which can induce conflicts on the output port as the traffic from a total of 47 input ports is output to a single output port.

Figure 5.14 shows the resulting latency distribution of the foreground flow when (a) enabling quality of service features for the foreground flow or (b) without any special settings for the flow on the switch. When the quality of service feature is enabled in Figure 5.14 (a) we can clearly identify a bimodal distribution showing that there are two internal paths the traffic can take in the switch. If the output ports happens to be unused when a packet of the foreground flow arrives, the packet gets forwarded in cut-through logic, i.e., immediately. The peak of the latency distribution for this cut-through operation is 921.6 ns. However, the switch might already be busy transmitting a packet of the 11.9 Mpps background flow in which case our packet needs to be forwarded in store-and-forward manner for a short time, yielding a second peak at 3.59 µs. The priority settings ensure that it does not get queued for too long to get a clearly visible bimodal distribution. Disabling the priority setting in Figure 5.14 (b) yields a



(a) Quality of service enabled for foreground traffic



(b) Quality of service disabled for foreground traffic

Figure 5.14: Latency distribution of a 1 Gbit/s flow forwarded by a hardware OpenFlow switch with 8 Gbit/s of background traffic [39]

long-tail distribution with latencies of up to 17µs instead because the queue length is unpredictable.

This test setup allows us to scale MoonGen beyond 100 Gbit/s of generated test traffic: Figure 5.15 shows latency results for both background and foreground traffic under increasing background traffic. The 46-fold amplification allows for a background flow of up to 414 Gbit/s (616 Mpps) while keeping the foreground traffic at 1 Gbit/s. Note that the constant offset between the minimum and median latencies of background and foreground traffic is because the background traffic passes through the switch twice due to the amplification (cf. Figure 5.13). For latency of an amplified background packet we define the latency as the first copy of the packet that is returned to MoonGen. The key result of this measurement is the maximum latency, clearly showing that the quality of service feature in the switch is working as intended, even before the output port is overloaded. Once the output port is overloaded (background traffic $\geq$ 16 Gbit/s in the measurement) the bimodal distribution disappears as all packets are queued since the

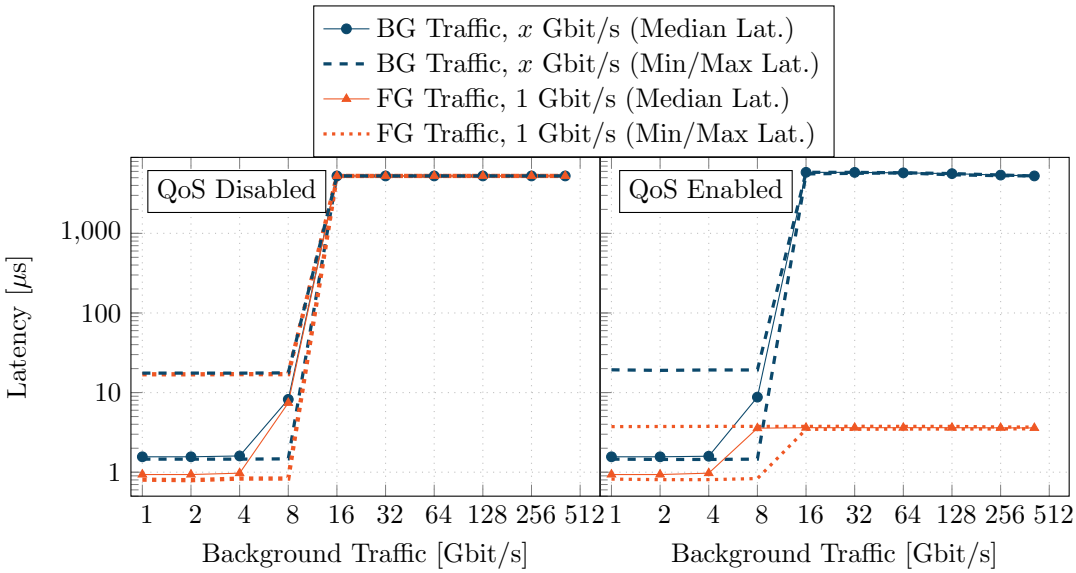


Figure 5.15: Background (BG) and foreground (FG) flow latency under increasing background load [39])

output port is always occupied by background traffic when a packet of the foreground flow arrives.

It is not a goal of this dissertation to analyze the internal details of an OpenFlow hardware implementation, interested readers are referred to our publication about traffic amplification and benchmarking this OpenFlow switch with MoonGen from which this measurement was taken [39]. Instead, this example is meant as a practical demonstration of MoonGen's capabilities: We can characterize complex behavior of a hardware switch under a total load of 415 Gbit/s with 618 Mpps (Figure 5.15). Our timestamping precision is high enough to clearly identify probability distributions in a device that takes less than a microsecond to forward a packet (Figure 5.14).

5.9 Conclusions

We have presented a general-purpose software packet generator that uses hardware features of commodity NICs to implement functionality that was previously only available on expensive special-purpose hardware. MoonGen represents a hybrid between a pure software-based solution and one based on hardware. It combines the advantages of both approaches while mitigating shortcomings by using both hardware-specific features and novel software approaches. MoonGen measures latencies with a precision of ±12.8 ns and a systematic error of only 317 ns (Section 5.6). The desired packet rate can be con-

trolled precisely through both hardware-support and our rate control algorithm based on filling gaps with invalid packets (Section 5.7).

Addressing question Q2 (Can high-level languages be used in software-based packet processing tools?), we use MoonGen as real world example of such a system. The Lua scripting languages gives users unprecedented flexibility and control over how MoonGen crafts packets: Users are not restricted to predetermined patterns but have full control and can test complex protocols. At the same time we achieve our performance goals from Section 2.2 of 14.88 Mpps on a single core (Section 5.5.2). This makes MoonGen both flexible and fast. The flexibility goes beyond the capabilities provided by hardware load generators as each packet can be crafted in real-time by a script. Tests that respond to incoming traffic in real-time are possible as MoonGen also features packet reception and analysis (Section 5.4).

The key takeaway from this chapter is that MoonGen is fast, flexible, and features precise timestamping that is even precise enough to characterize hardware devices (Section 5.8). This makes MoonGen the answer to research question Q3: How can modern network devices be benchmarked? We recommend to use MoonGen with an Intel 82599 NIC with fiber optic transceivers to achieve the stated precision and accuracy. All software packet generators prior to MoonGen had serious shortcomings (lack of hardware timestamping and poor configurability) and were unsuitable to run the experiments required for this dissertation.

5.10 Author's Contributions

Sections 5.1 to 5.9 are based on the following publication [42]:

> Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. "MoonGen: A Scriptable High-Speed Packet Generator". In: *Internet Measurement Conference 2015 (IMC'15)*. Tokyo, Japan, Oct. 2015

The author of this thesis architected and implemented the core of the MoonGen packet generator. All benchmark experiments were conceived and conducted by the author. The evaluation of hardware timestamping and clock synchronization was done by the author. The evaluation of different rate control methods was done by the author. All analyses and conclusions drawn from experiments was the author's work.

The following significant changes vs. the paper were made for this thesis:

- Measurements about rate control precision have been removed as they have been obsoleted by a later publication that is discussed in the next chapter [41].

- Hardware support and feature discussions have been updated throughout the text to match recent developments in MoonGen.
- Example code was updated to match recent changes to MoonGen.
- The discussion about the choice of the Lua programming language and DPDK backend has been updated.
- The discussion of clock drift contains a new measurement (Figure 5.9) and system overview diagram (Figure 5.8).
- Section 5.6 has been updated: Figures 5.6 and 5.7 are new. The explanation of the timestamping experiment has been extended. A mistake in the calculation of the error bars for the propagation delay and constant offset has been corrected.
- Section 5.8 is new (measurements based on [39]).

Chapter 6

Precision of Software-based Packet Generators

Building high precision benchmarking tools in software in a high-level language running on commodity hardware is a difficult task. We need to quantify the precision of timestamping and rate control in MoonGen to validate in which scenarios a software solution can replace a hardware solution (research question Q4).

The goal of this chapter is to validate that MoonGen's hybrid solution to this problem works better than the state of the art in software. We compare our implementation to a variety of different software packet generators. We also quantify the limits of different methods for rate control and timestamping gain a deeper understanding about the trade-offs involved. In order to achieve these goals we need a more precise measurement device as ground truth: We use OSNT [7], hardware packet generator based on the NetFPGA platform to this end.

The remainder of this chapter is based on our publication about the precision packet generators which is joint work by Paul Emmerich, Sebastian Gallenmüller, Gianni Antichi, Andrew W. Moore, and Georg Carle. [41]. A full account of the author's contributions is given in Section 6.9.

6.1 Introduction

Software packet generators traditionally relied on the operating system's network stack and implemented all timing-critical parts in the user space process itself. This introduces uncertainties caused by the operating system's scheduling and processing time in the network stack. The closer you can move packet scheduling and latency measurements

to the actual physical transmission of the packet on hardware, the better. Clearly, a pure hardware solution is ideal for these aspects of a packet generator. The question is how close to we need to get with a pure software approach and how close can we get?

We start with motivating examples in Section 6.3 that show how precision of the packet generator can influence the behavior of the system under test and skew the measurement results. Our evaluation in Section 6.6 compares different ways to control the inter-departure time of packets to validate MoonGen's novel solution. Section 6.7 focuses on latency measurements and shows the shortcomings of purely software-based solutions.

6.2 Related Work

The performance and precision of software packet generators has been subject of previous studies. In particular, Paredes-Farrera et al. [128] 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. [23] 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 [148, 42]. Bonelli et al. [20] 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.

To this end, we performed our measurement campaign using OSNT [7]: an open source hardware based traffic generator and monitoring system based on the NetFPGA [170]. 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.

6.3 Precision of packet generators affects measurement results

The behavior of a system under test can change in significant ways when exposed to a packet generator that fails to adhere to the configured traffic pattern. Traffic pattern refers to the distribution of inter-packet gaps. For example, sending an average rate of 1 Mpps can be done in different ways: by sending one packet every microsecond or by sending 100 packets in bursts every 100 microseconds. Both traffic looks the same to an unsophisticated observer at timescales typically observed by statistics outputs (i.e., seconds) but very different to a low-level system.

6.3.1 Generating CBR traffic

One example of a low-level system that interprets these traffic patterns differently are the interrupt throttling mechanisms found in drivers and operating systems. Figure 6.1 compares the number of interrupts per second observed on a system forwarding packets with Open vSwitch when tested with load generated by MoonGen and the zsend packet generator (based on PF_RING ZC [135]). We use the interrupt rate as a white-box measurement as an example of complex system behavior.

Both packet generators were configured to send CBR traffic as defined by RFC 1242, i.e., a constant gap between individual packets [24]. This is a common configuration for benchmarking tests, for example, the RFC 2544 benchmarking standard requires this traffic pattern [25]. MoonGen was configured to use the hardware rate control on an Intel 82599 NIC (which sends CBR traffic according to the datasheet [73]), zsend was configured with the dedicated timing thread and CBR traffic. So both should generate the same traffic and the experiment should show the same behavior of the system under test. However, white-box measurements show a completely different behavior on the system under test. When measuring the interrupt rate on the system under test we observe a large difference while forwarding packets under low load (Figure 6.1).

This indicates that a significant difference in the behavior of a system when observed in two supposedly identical experimental setups (both test tools were configured identically). As it turns out neither packet generator sends CBR traffic in this configuration, even the hardware implementation on the Intel NIC is flawed. The problem here is batching which is required for high performance in software packet processing. Batching packets leads to bursts on the wire, resulting in traffic patterns significantly different from the expected CBR traffic. zsend sends large bursts instead of the configured CBR traffic due to a bug in its implementation, the Intel NIC sends bursts of size 2 for unknown reasons. A full evaluation follows in Section 6.6.

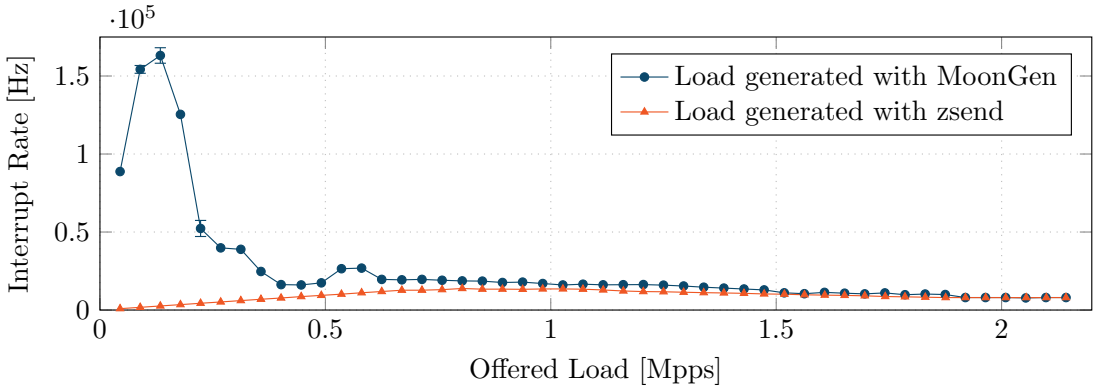


Figure 6.1: Observed interrupt rate when forwarding packets with Open vSwitch, traffic generated with different packet generators (Figure from [42])

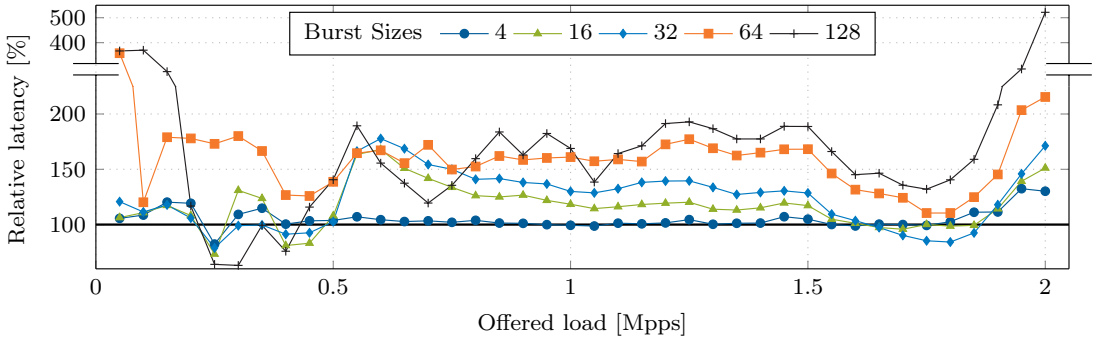


Figure 6.2: Relative observed latency of different burst sizes (traffic generated with MoonGen CRC rate control)

At first glance, this might seem like a curiosity of questionable practical relevance for the typically conducted black-box tests. However, there is a significant difference in externally observable behavior of the system: the median forwarding latency varies with the batch size by up to a factor of 3.7 between a burst size of 1 and a burst size of 128.

We run the same forwarding test with varying burst sizes and measure the observed latency. Figure 6.2 compares true CBR traffic with explicitly configured burst traffic by plotting the median latency. We use the MoonGen packet generator with the CRC rate control method (cf. Section 5.7.3 in Chapter 5) to precisely generate the burst sizes. Latency are measured using hardware timestamping as previously evaluated in Section 5.6 in Chapter 5. The latency is expressed as relative with respect to the CBR case.

If burst sizes would not make a difference, then the corresponding plots in Figure 6.2 would simply be a flat line at 100%. However, we observe large differences of up to 370%. Bursts of 4 packets already show a large impact on the observed latency: the maximum deviation in the non-overload cases is at 2.0 Mpps. We observe a latency of 21.2 µs with burst size 4, but only 16.3µs with true CBR traffic, a 30% difference.

This figure demonstrates that a system under test can show 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 as the measurements show a complex behavior. 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 configure batch sizes between 16 and 512 packets resulting in bursts of equivalent sizes. This is an intentional setting to improve performance. 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.

6.3.2 Generating Poisson Traffic

CBR traffic is often an unrealistic test scenario for measurements of latency. However, CBR or very large bursts are often the only options offered by packet generators. A more realistic setup could use a Poisson process to also stress buffers as the system under test becomes temporarily overloaded.

Figure 6.3 shows latencies of the Open vSwitch test setup comparing CBR with Poisson traffic. Note that the maximum achieved throughput of about 1.9 Mpps is the same regardless of the traffic pattern. However, the behavior at lower latencies differs between these two traffic patterns. The (perfectly reproducible) outlier at 0.4 Mpps for CBR traffic are likely artifacts of the interaction between the interrupt throttle algorithm found in the Intel driver [70] and the dynamic interrupt adaption of Linux in NAPI [150]

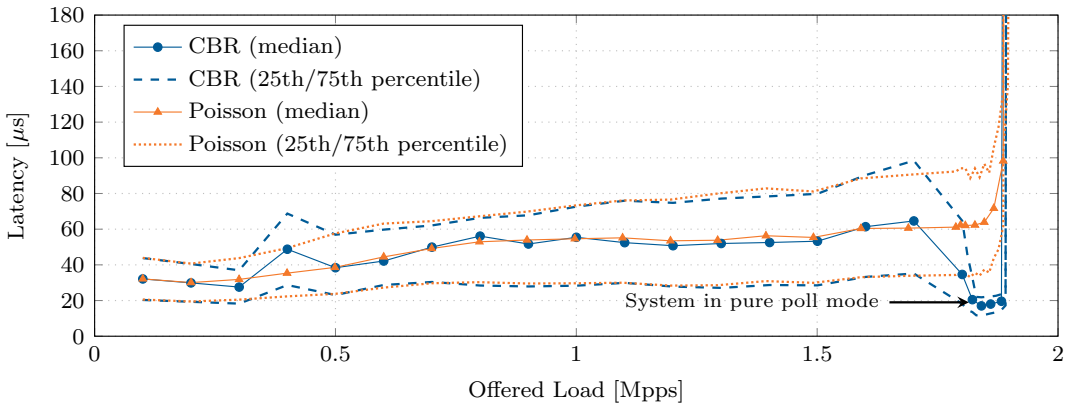


Figure 6.3: Forwarding latency of Open vSwitch with CBR and Poisson traffic patterns (Figure from [42])

on the system under test. The sudden drop in latency before hitting the maximum load is because the perfect CBR traffic puts interrupt into permanent polling mode, eliminating delays from interrupt throttling and idle states. More information on the interactions between interrupt throttling and dynamic polling in NAPI can be found in our publication about modeling latency in the Linux kernel [16].

A Poisson process generates a smoother result and is arguably a more realistic test setup as it resembles real-world traffic more closely. Yet, most software traffic generators do not offer this traffic pattern. We also show that generating Poisson traffic is actually *easier* for software packet generators in a reliable way in this chapter to argue that mosts benchmarks should be run with this traffic pattern instead of CBR. Researchers might also find value in comparing different traffic patterns to gain insights into internals of black-box devices.

6.4 State of the Art for Software Packet Generators

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 [128],

while the latter is caused by the network stack itself which is optimized for compatibility and stability rather than high performance or high precision [148, 18].

We investigate 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 [32] and `trafgen` [162] 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 we choose D-ITG because of its ability to generate both realistic and synthetic traffic patterns [22] 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 when used with a selection of high-speed network cards. 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 [53] 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. [23, 128]. 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.

Selected packet generator include Windriver System's Pktgen-DPDK [136] and MoonGen [3]. 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 [111] is a widely used packet generator included in netmap while PFQ offers pfq-gen [133]. PF_RING ZC [135], a high-performance framework found in production systems provided by ntop [116], offers the packet generator zsend. Table 6.1 summarizes the investigated software packet generators.

	Version	IO API
D-ITG [32]	v2.8.1	Linux
`trafgen` [162]	v0.5.7	Linux
Pktgen-DPDK [136]	v2.8.0	DPDK (v1.8.0)
MoonGen [3]	git 5cf96c72*	DPDK (v1.8.0)
	git bfe8b5b1†	
	git 39e0cb64‡	
pkt-gen [111]	git b24fce99	netmap
pfq-gen [133]	v5.2.9	PFQ
zsend [135]	v6.3.0.160209	PF_RING ZC

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

Table 6.1: Investigated software packet generators

6.5 Test setup

Our main test setup consists of two machines, directly connected via two 10 Gbit/s fibre links. One is equipped with a NetFPGA-10G [110] programmed with OSNT [7] for high precision packet inter-arrival time characterization. OSNT allows for nanosecond-precision timestamping of all received packets in real time at full line rate with minimum sized packets [7] and acts a full hardware implementation to verify our software implementations.

The other system is used to run the software packet generator and 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. Note that this test setup is different from the other setups (Section 2.4) used in this dissertation as these experiments were conducted at a lab in the University of Cambridge where the necessary hardware was available.

6.6 Analysis of rate control

Rate control is the mechanism implemented by a traffic generator to assure the generated traffic matches the required characteristics. We assess the following criteria for rate control of the selected packet generators:

- **Bandwidth:** the maximum throughput (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.

6.6.1 Rate control: three different approaches

Software packet generators can generally use three different ways for rate control: relying on hardware features of NICs, using a pure software approach, or MoonGen's approach with corrupted packets called CRC rate control. See Chapter 5 Section 5.7 for a detailed explanation how these work. We implemented all of these methods in MoonGen to compare them in a fair manner.

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 [128, 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. All other evaluated packet generators support only this method.

MoonGen's CRC rate control works by injecting invalid packets between real ones to adjust the desired inter-packet gap. 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.

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 6.2: Achieved throughput on a Core i7-960

Our hardware-assisted solution relies on capabilities of the Intel 82599 [73] and Intel XL710 [77] NICs to do rate control in hardware.

6.6.2 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 (cf. Section 3.5.3). This leads to a trade-off between speed and precision for packet generators when the user does not require bursty distributions. Indeed, all frameworks mentioned in Section 3.2 for high-speed packet IO use large batch sizes by default: Table 6.2 shows the defaults for the investigated packet generators. These defaults are often unsuitable for precise packet generation due to the large batch size.

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 6.2 summarizes the 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 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 [148], 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/s links. D-ITG reported incorrect statistics at rates above 0.1 Mpps which is only 67 Mbit/s with minimum-sized packets and 1.2 Gbit/s with maximum-sized packets.

6.6.3 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 6.3 shows measurement results and relative errors. `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 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 6.3. 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.

6.6.4 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 of above 6 Mpps (cf. Section 6.6.2), we use a packet size of 128 bytes (including CRC) corresponding to a maxi-

Packet generator	Target [Mpps]	Measured [Mpps]	Rel. error
D-ITG	0.01	0.01	< 0.1%
	0.1	0.099	0.6%
	0.2	0.15	75%
`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%
MoonGen (CRC)	1	1.00	< 0.1%
	3	3.00	< 0.1%
	5	4.99	0.2%
MoonGen (SW)	1	1.00	< 0.1%
	3	3.00	0.1%
	5	5.00	< 0.1%
zsend	1	1.00	< 0.1%
	5	5.00	0.1%
	8	8.00	0.1%
Pktgen-DPDK	0.976	0.840	16%
	2.54	2.17	17%
	4.1	3.45	19%
pfq-gen	1	1.00	< 0.1%
	2	2.00	< 0.1%
	3	3.00	< 0.1%
netmap pkt-gen	1	1.00	< 0.1%

Table 6.3: Accuracy evaluation

mum packet rate of 8.45 Mpps for the evaluation. Packet size typically does not influence the performance of packet generation [148]. 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.

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 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 clocking 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 6.6.3. 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.

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

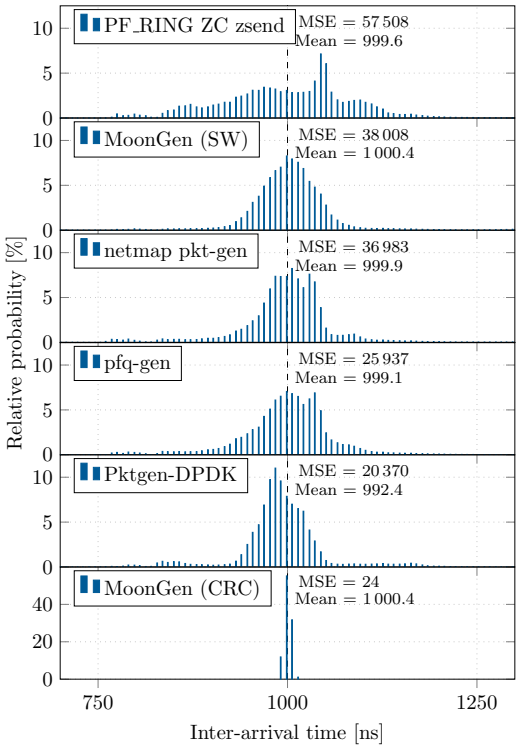


Figure 6.4: Software rate control at 1 Mpps (1000 ns inter-arrival time) with mean squared error (MSE) per packet generator

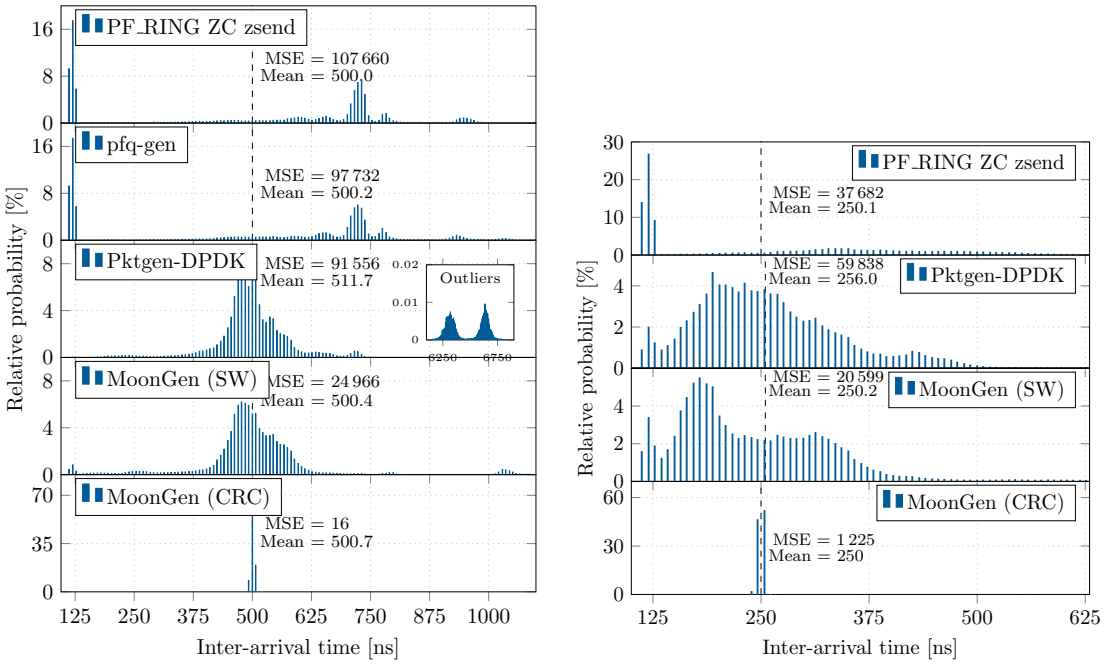
`ktime_get_real()` in a busy-wait loop. Note that this function does not use the RDTSC instruction to determine time on each call, limiting the precision.

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.

Results at 1 Mpps

Figure 6.4 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



(a) 2 Mpps, 500 ns target

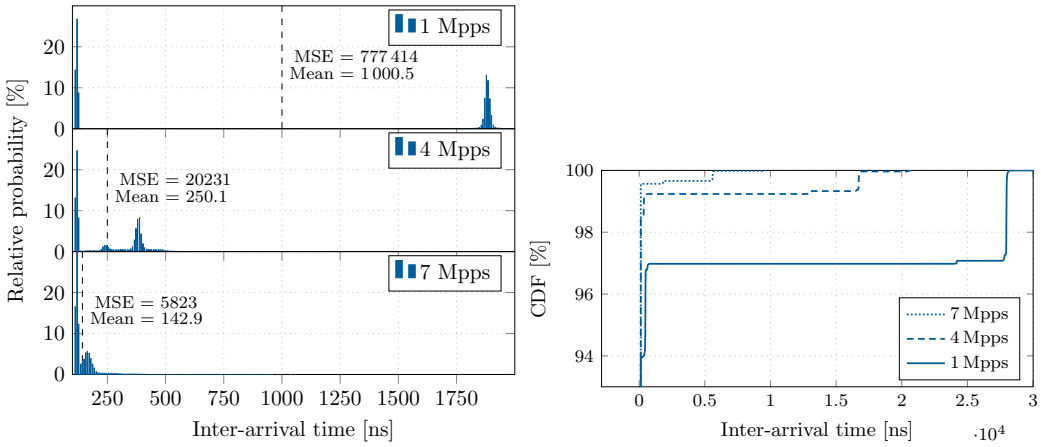
(b) 4 Mpps, 250 ns target

Figure 6.5: Software rate control at 2 Mpps and 4 Mpps

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 resulting 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.

Results at higher rates

Figures 6.5a and 6.5b 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



(a) 82599 NIC (b) XL 710 NIC

Figure 6.6: MoonGen with hardware rate control at rates of 1, 4, and 7 Mpps

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. These outliers happen periodically every second. 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 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.

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 [42]. However, this evaluation was restricted to 1 Gbit/s due to restrictions of the measurement setup. Here, we use MoonGen to evaluate NICs based on Intel 82599 and XL710 chips which feature SFP+ ports (10 Gbit/s).

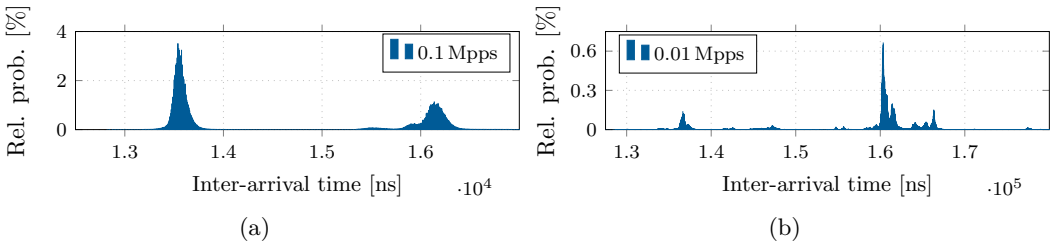


Figure 6.7: trafgen precision

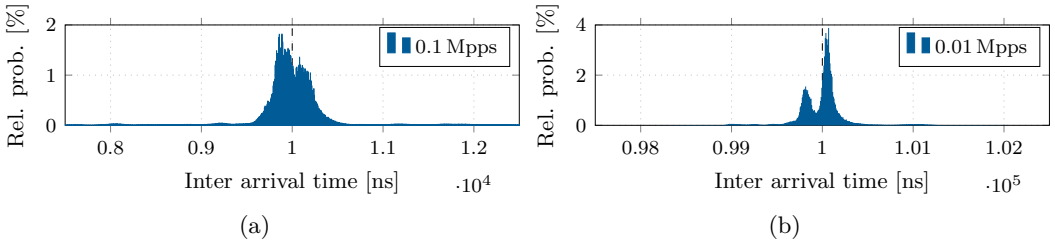


Figure 6.8: D-ITG precision

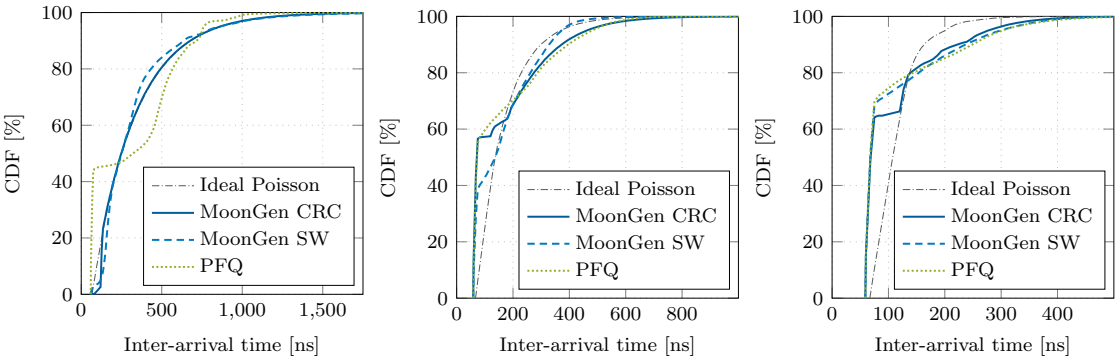
The Intel 82599 datasheet outlines the rate control algorithm in detail [73, Section 7.7.2.1]. Figure 6.6a shows that the hardware generates bursts of two packets at rates of 1, 4, and 7 Mpps on Intel 82599 NICs. The algorithm described in the datasheet does not match this observation, highlighting problems 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 6.6b shows the upper part of a CDF as these large bursts are not easily visualized in a histogram.

A burst size of 2 has no significant effect on the system under test in our experience. Measuring the latency of a system with the flawed hardware rate control and MoonGen's CRC approach results in the same latency [42]. So we find it good enough for practical purposes. Rate control on the XL710, however, is flawed and should be avoided.

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 Mpps and 0.1 Mpps (corresponding to 11.8 Mbit/s to 112 Mbit/s). Figure 6.7 and 6.8 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.



(a) 3 Mpps (1 thread) (b) 6 Mpps (2 threads) (c) 9 Mpps (3 threads)

Figure 6.9: Precision of Poisson traffic generation

D-ITG generates a bimodal distribution oscillating around the target rate, alternating every ≈ 2 packets. The two modes of the distribution are only ≈ 200 ns away from the target at both tested rates. Note that an error of 200 ns is good result at 1 Gbit/s because the minimum packet length is 672 ns at this rate. It is, however, prone to bursts as the rate increases. 0.5% of the 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.

6.6.5 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 [129], traffic generated by a Poisson process can approximate the self-similar pattern for short time spans, e.g., in a synthetic benchmark [149]. 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 im-

prove 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 6.9 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 test device was receiving such packets. This test was therefore run with packet sizes of 64 bytes or higher, MoonGen's actual precision is likely higher.

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. And this is were we hit the limits of what can be done in software, there is no way to avoid this without explicit hardware support.

6.6.6 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 6.3). 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.

6.7 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 system 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 and received.

6.7.1 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 take a timestamp at the moment the NIC is tasked with transmitting the packet. Reception typically works in a busy-wait loop, polling the NIC for new packets as often as every 100 CPU cycles, a timestamp can be taken once a poll returns data. The question is: how precise is this?

The 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). Typical values for timestamping precision on 10 Gbit/s NICs are ±6.4 ns, see Section 5.6.1 for details.

6.7.2 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 system 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 [73], older 1 Gbit/s NICs often support only 64 ns [72].

6.7.3 Evaluation

In this section we evaluate the three different approaches using an external loopback connection of a NIC port with itself via a short ($\approx$ 10 cm/2.5 in) fiber cable as test setup.

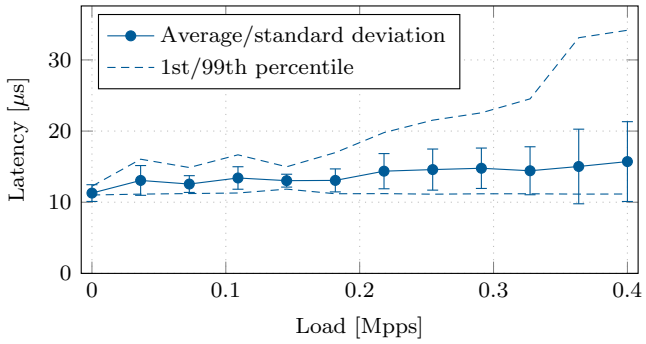


Figure 6.10: Precision of software timestamping without framework support

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.

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.

Software without framework support

Figure 6.10 shows how the precision of software timestamping without fast IO framework support deteriorates as we apply an increasing load of background traffic using `trafgen`[1]. The standard deviation is in the range of several microseconds (vs. nanoseconds with hardware timestamping) and the systematic error is 10 µs to 15 µs as this is the absolute measured value.

Software with framework support

Software timestamping with DPDK support in MoonGen performs better by an order of magnitude as visualized in Figure 6.11. 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 µs. High background traffic causes this software

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

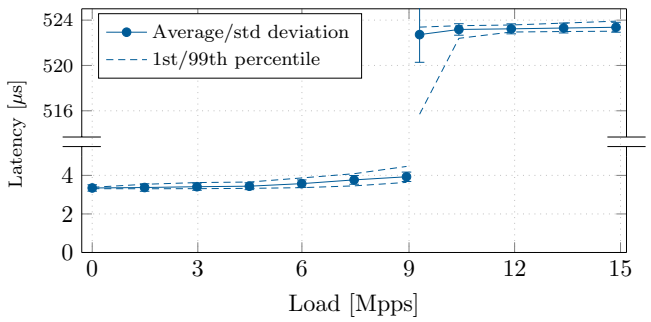


Figure 6.11: Precision of software timestamping with framework support (MoonGen)

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, but is likely related to the hardware rate control feature being used an buffering on the NIC. Note that 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 as the buffers are always full.

6.7.4 Lessons learned

The test campaign performed in this section assess the precision of software timestamping with and without IO framework support. 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 6.3 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.

6.8 Conclusion

This chapter is mainly concerned with research questions Q3 (How can modern network devices be benchmarked?) and Q4 (Can software be sufficiently precise to replace hardware in all scenarios?). The answer is: mostly yes, software can be precise enough to replace dedicated hardware for benchmarking in 10 Gbit/s networks. Only timestamping is best left to hardware features, but MoonGen's driver support for hardware timestamping on commonly used NICs makes these capabilities available on cheap commodity off-the-shelf hardware.

The results for rate control are unexpected. The hardware implementation on the Intel XL710 NIC behaves very poorly and is unsuitable for packet generators. Intel's 82599-based NICs fared better, but still sent out packets in bursts of two which we consider good enough for practical purposes. MoonGen using the rate control implemented with the corrupted CRC approach proved to be the most precise solution. Our conclusion is that you should use MoonGen's CRC approach (Section 5.7.3) to control packet rate if the system under test can handle this. We achieve a mean squared error of 1.2 µs with this implementation, significantly better than the 20.6 µs of the software implementation, which in turn is still an improvement over the 59 µs found in other

software packet generators (Section 6.6.4). The next best option is hardware rate control on NICs of the 82599 family. Software rate control should be avoided.

Another key result is that the traffic pattern should be chosen carefully. Benchmarking standards are often focused on CBR traffic, a fixation that arguably stems from the definition of constant load as CBR in RFC 1242 [24]. Even this standard from 1991 notes that it is an unrealistic traffic pattern. 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 (Section 6.3.2).

In conclusion, we can now say that MoonGen is a precise and accurate software packet generator when used with Intel 82599 10 Gbit/s NICs.

6.9 Author's Contributions

Sections 6.1 to 6.8 are based on the following publication [41]:

> Paul Emmerich, Sebastian Gallenmüller, Gianni Antichi, Andrew W. Moore, and Georg Carle. "Mind the Gap — A Comparison of Software Packet Generators". In: *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2017)*. Beijing, China, May 2017

The author implemented the different rate control methods of MoonGen discussed here and contributed to their analysis. The author contributed to the design and execution of all measurements and experiments of the rate control methods presented here. All timestamping code in MoonGen evaluated here was conceived and implemented by the author. The author contributed to the experiments and their execution.

The following significant changes vs. the paper were made for this thesis:

- Section 6.1 has been rewritten.
- Section 6.3 has been rewritten and features two additional experiments originally published in [42].
- Table 6.3 has been updated to contain data about all evaluated packet generators.

Part IV

Conclusion

Chapter 7

Conclusion

We describe the software stack of a software-based packet processing system, starting at the driver level (ixy, Chapter 3) through to a real-world application (MoonGen, Chapter 5). At each level we evaluate the performance and trade-offs between flexibility and performance, yielding unique insights at every level of the stack. Having a completely custom driver allowed us to evaluate effects of individual features and optimizations in isolation. Doing the same in ten different languages made for a comprehensive comparison of language features like garbage collection and their performance impacts (Chapter 4).

Finally, as a complex, real-world, application MoonGen is a practical example of a fast and flexible software-based packet processing system. The combination of fast user space drivers together with a high-level language can provide completely new functionality that was previously impossible: running user-defined script code for every single packet that is transmitted without affecting performance (Section 5.5.2).

7.1 Answered research questions

We can now offer answers to our four main research questions posed in Section 1.2.

Q1 What makes software-based packet processing fast?

Being close to the hardware is crucial: Unnecessary layers of abstraction such garbage collection and JIT compilers may add latency, jitter, or performance problems and hence must be avoided (Section 4.6 and 4.7). The driver should be part of the same process as the application, i.e., the driver must be a library that is embedded in the user space application (Section 3.3). Chapter 3 examines the

architecture of fast user space network drivers, featuring a full implementation of a driver that rivals the performance of DPDK (Section 3.5).

We consider important components of user space drivers and discuss how different settings affect the performance. The single most important identified performance feature is batch processing. Loading at least 32 packets at the same time helps to amortize overhead related to accessing the hardware (Section 3.5.3). Moreover, we look at the necessary trade-offs, identifying a ring buffer size 512 as the sweet spot between throughput and latency (cf. Section 3.5.7). This is contrary to common advice for larger batch sizes and ring sizes (e.g., [148, 89, 12]).

Q2 How can high-level languages be used in software-based network devices?

Not all high-level languages are suitable for writing drivers, especially scripting languages come with serious performance bottlenecks. We implemented drivers in 9 different high-level languages in Chapter 4 to evaluate and compare them.

Our results indicate that Rust is currently the best language to write user space drivers in, winning performance comparisons for both latency and bandwidth among the high-level languages. Compared to C we achieve 98% of the throughput (Section 4.6.2) and without affecting latency at speeds of 10 Mpps (Section 4.7.2). Despite achieving a high performance we guarantee memory safety in 87% of our Rust driver (Section 4.4.3). This is not entirely unexpected given that Rust was explicitly designed with writing efficient and correct applications in mind.

All other languages have one primary problem: overhead from memory management. Relying on a garbage collector yields a bad worst-case latency when an ongoing garbage collection stops execution temporarily (see graphs in Section 4.7). The garbage collected languages C# and Go fared exceptionally well for garbage collected languages, achieving worst-case latencies below 100 µs even under load. Even Apple's Swift that relies on pure reference counting fares poorly in this comparison: the overhead heavily affects the throughput as unnecessary work has to be done for every packet. Only Rust's unique ownership-based system avoids all of these problems.

On the higher layer, our packet generation application MoonGen uses the scripting language Lua, a garbage collected language. Avoiding problems with garbage collection is feasible here by clever use of available hardware features where a high precision is required (Section 5.6). Lua is an ideal choice for a testing application: it is easy to fully customize the behavior of the packet generator by the user on a per-packet basis (Section 5.4). Our evaluation shows that we achieve equivalent, or better, performance than previous less flexible packet generators (Section 5.5.2).

We conclude that Rust is the best choice for general-purpose software-based packet processing systems. Other languages can be feasible if the pause times induced by garbage collection or JIT compilers can be compensated for by hardware features, as is the case for MoonGen.

Q3 How can modern network devices be benchmarked?

Our focus is on throughput on packets per second, not raw bandwidth in bits per second as the per-packet processing costs dominates over per-byte costs (Section 2.2). We built the packet generator MoonGen for this dissertation – the first software packet generator that allows running user-defined script code for every single generated packet without sacrificing performance (Section 5.5.2). This allows supporting a wide range of protocols and applications (Section 5.4).

Despite being a software-based packet processing system, MoonGen achieves a precision of ±12.8 ns for timestamping by using hardware features of commodity NICs (Section 5.6).

We conclude that MoonGen is a packet generator suitable for accurate and precise benchmarks in 10 Gbit/s networks. It should be used with an Intel 82599 NIC in order to use hardware features to achieve the previously stated results. We show MoonGen is precise enough to characterize hardware switches as devices under test (Section 5.8).

Q4 Can software be sufficiently precise to replace hardware in all scenarios?

There are scenarios where hardware support is inevitable: Everything that either requires timestamping or precise control of packet departure times (rate control). This is a rare use case, many applications simply do not need these capabilities. And when needed, a hybrid solution such as MoonGen can be constructed to work around the restrictions.

MoonGen achieves a mean-squared error of 1.2 µs by using a hybrid hardware/-software rate control implementation, the previous state of the art for software packet generators was 59 µs (Section 6.6.4). Our hardware-assisted timestamping achieves a precision of ±12.8 ns and a systematic error of 317 ns (Section 5.6).

We find these values good enough for practical purposes in 10 Gbit/s networks.

7.2 Conclusion

This dissertation heavily relies on the artifacts ixy and MoonGen to argue its main point: It is feasible and advantageous to write packet processing systems in high-level languages on top of user space drivers.

The advantage of using high-level languages is improved resilience against certain classes of bugs such as memory bugs. 61% of security-critical bugs found in Linux could have been prevented by using a high-level programming language instead of C and all but one investigated bug (n=40) have been found in drivers (Section 4.3.2). When using Rust we get this advantage in 87% of the code while only trading off 2% of the throughput and no additional latency under normal load conditions (Sections 4.6.2 and 4.7.2). We conclude that software-based packet processing systems should be build in a high-level language. Language selection depends on requirements, but Rust is a good choice in general.

MoonGen is a practical example of a fast and flexible software-based packet processing system. It is fast, flexible, and precise at the same time by using a high-level language and hardware features to achieve precision. We achieve our goal of handling the full line rate of 14.88 Mpps on a single CPU core (Section 5.5), flexibility is ensured by executing user-defined script code in the high-level language Lua for every transmitted packet (Section 5.4), and the achieved precision for timestamping of ± 12.8 ns is even good enough to benchmark hardware devices (Section 5.8).

It has been a personal goal of the author to build something that can be adapted and re-used by others. Both ixy and MoonGen have been made available as free and open source software [2, 3]. This has been especially successful with MoonGen which has become the de-facto standard packet generator in academia ($\geq$ 300 citations) and it has been used in many high-impact publications by others, e.g., [152, 93, 34, 168].

Bibliography

[1] Dave Abrahams. "Protocol-Oriented Programming in Swift". In: *WWDC15* (June 2015).

[2] Paul Emmerich et al. *ixy code*. `https://github.com/emmericp/ixy`.

[3] Paul Emmerich et al. *MoonGen Source Code*. `https://github.com/emmericp/MoonGen`.

[4] Alexander Frank. "Cost Efficient Hardware Timestamping". In: *TUM Interdisciplinary Project Final Report*. TUM. 2018.

[5] Alexander Frank. *MoonSniff*. `https://github.com/emmericp/MoonGen/pull/227`. 2018.

[6] Anders Evenrud et al. *OS.js*. `https://www.os-js.org`. 2019.

[7] Gianni Antichi, Muzammil Shahbaz, Yiwen Geng, Noa Zilberman, Adam Covington, Marc Bruyere, Nick McKeown, Nick Feamster, Bob Felderman, Michaela Blott, et al. "OSNT: Open Source Network Tester". In: *Network* 28.5 (2014).

[8] Serhat Arslan, Stephen Ibanez, Alex Mallery, Changhoon Kim, and Nick McKeown. "NanoTransport: A Low-Latency, Programmable Transport Layer for NICs". In: *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*. 2021, pp. 13–26.

[9] "AS5712-54X Datasheet". In: Edge-Core Networks. 2015.

[10] Hirochika Asai and Yasuhiro Ohara. "Poptrie: A Compressed Trie with Population Count for Fast and Scalable Software IP Routing Table Lookup". In: *ACM SIGCOMM Computer Communication Review* 45.4 (2015), pp. 57–70.

[11] Austin Clements, Rick Hudson. *Proposal: Eliminate STW stack re-scanning*. `https://go.googlesource.com/proposal/+/master/design/17503-eliminate-rescan.md`. 2016.

[12] Jamie Bainbridge and Jon Maxwell. "Red Hat Enterprise Linux Network Performance Tuning Guide". In: *Red Hat Documentation* (Mar. 2015). Available at `https://access.redhat.com/sites/default/files/attachments/20150325_network_performance_tuning.pdf`.

[13] Fred Baker. *RFC 1812: Requirements for IP version 4 routers*. 1995.

[14] David Barach, Leonardo Linguaglossa, Damjan Marion, Pierre Pfister, Salvatore Pontarelli, and Dario Rossi. "High-Speed Software Data Plane via Vectorized Packet Processing". In: *IEEE Communications Magazine* 56.12 (2018), pp. 97–103.

[15] Tom Barbette, Cyril Soldani, and Laurent Mathy. "Fast userspace packet processing". In: *Architectures for Networking and Communications Systems (ANCS)*. ACM. 2015, pp. 5–16.

[16] Alexander Beifuß, Daniel Raumer, Paul Emmerich, Torsten M Runge, Florian Wohlfart, Bernd E Wolfinger, and Georg Carle. "A Study of Networking Software Induced Latency". In: *2015 International Conference and Workshops on Networked Systems (NetSys)*. IEEE. 2015, pp. 1–8.

[17] Gilberto Bertin. *Single RX queue kernel bypass in Netmap for high packet rate networking*. `https://blog.cloudflare.com/single-rx-queue-kernel-bypass-with-netmap/`. Oct. 2015.

[18] N. Bonelli, S. Giordano, and G. Procissi. "Network Traffic Processing With PFQ". In: *IEEE Journal on Selected Areas in Communications* 34.6 (June 2016), pp. 1819–1833. ISSN: 0733-8716.

[19] Nicola Bonelli, Andrea Di Pietro, Stefano Giordano, and Gregorio Procissi. "Flexible High Performance Traffic Generation on Commodity Multi–Core Platforms". In: *Proceedings of the 4th International Conference on Traffic Monitoring and Analysis*. Vienna, Austria: Springer, 2012, pp. 157–170. ISBN: 978-3-642-28533-2.

[20] Nicola Bonelli, Andrea Di Pietro, Stefano Giordano, and Gregorio Procissi. "Flexible High Performance Traffic Generation on Commodity Multi–Core Platforms". In: *International Conference on Traffic Monitoring and Analysis (TMA)*. Springer, 2012.

[21] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. "P4: Programming protocol-independent packet processors". In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014), pp. 87–95.

[22] Alessio Botta, Alberto Dainotti, and Antonio Pescapè. "A tool for the generation of realistic network workload for emerging networking scenarios". In: *Computer Networks* 56.15 (2012).

[23] Alessio Botta, Alberto Dainotti, and Antonio Pescapé. "Do You Trust Your Software-Based Traffic Generator?" In: *IEEE Communications Magazine* 48.9 (2010), pp. 158–165.

[24] Scott Bradner. *Benchmarking Terminology for Network Interconnection Devices*. RFC 1242 (Informational). Internet Engineering Task Force, 1991.

[25] Scott Bradner and Jim McQuaid. *Benchmarking Methodology for Network Interconnect Devices*. RFC 2544 (Informational). Internet Engineering Task Force, 1999.

[26] Stuart Cheshire, David Schinazi, and Christoph Paasch. "Advances in Networking". In: *WWDC17* (June 2017).

[27] Cilium Project. *BPF and XDP Reference Guide*. `https://docs.cilium.io/en/latest/bpf/`. 2021.

[28] Cloudflare. *quice: Savoury implementation of the QUIC transport protocol*. `https://github.com/cloudflare/quiche`. 2019.

[29] G. Adam Covington, Glen Gibb, John W. Lockwood, and Nick McKeown. "A Packet Generator on the NetFPGA Platform". In: *17th IEEE Symposium on Field Programmable Custom Computing Machines*. 2009, pp. 235–238.

[30] Daniel Crevier. *AI: The Tumultuous Search for Artificial Intelligence*. BasicBooks, 1993. ISBN: 0465029973.

[31] Cody Cutler, M Frans Kaashoek, and Robert T Morris. "The benefits and costs of writing a POSIX kernel in a high-level language". In: *OSDI'18*. USENIX. 2018, pp. 89–105.

[32] *D-ITG*. `http://traffic.comics.unina.it/software/ITG/index.php`.

[33] Dan Williams. *Solo5 project*. `https://github.com/Solo5/solo5`. 2019.

[34] Huynh Tu Dang, Han Wang, Theo Jepsen, Gordon Brebner, Changhoon Kim, Jennifer Rexford, Robert Soulé, and Hakim Weatherspoon. "Whippersnapper: A P4 Language Benchmark Suite". In: *Proceedings of the Symposium on SDN Research*. 2017, pp. 95–101.

[35] DPDK Project. *DPDK: Supported NICs*. `http://dpdk.org/doc/nics`.

[36] DPDK Project. *DPDK User Guide: Overview of Networking Drivers*. `http://dpdk.org/doc/guides/nics/overview.html`.

[37] DPDK Project. *Vhost-user CVE-2018-1059*. Mailing list post. `http://mails.dpdk.org/archives/announce/2018-April/000192.html`. 2018.

[38] Paul Emmerich. *MoonGen Source Code, Commit 96818ad9*. `https://github.com/emmericp/MoonGen/commit/96818ad962db92345674242c6935e413ba4eddcd`.

[39] Paul Emmerich, Sebastian Gallenmüller, and Georg Carle. "FLOWer – Device Benchmarking Beyond 100 Gbit/s". In: *IFIP Networking 2016*. Vienna, Austria, May 2016.

[40] Paul Emmerich, Daniel Raumer, Florian Wohlfart, and Georg Carle. "Assessing Soft- and Hardware Bottlenecks in PC-based Packet Forwarding Systems". In: *ICN 2015* (2015).

[41] Paul Emmerich, Sebastian Gallenmüller, Gianni Antichi, Andrew W. Moore, and Georg Carle. "Mind the Gap — A Comparison of Software Packet Generators".

In: *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2017)*. Beijing, China, May 2017.

[42] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. "MoonGen: A Scriptable High-Speed Packet Generator". In: *Internet Measurement Conference 2015 (IMC'15)*. Tokyo, Japan, Oct. 2015.

[43] Paul Emmerich, Simon Ellmann, Fabian Bonk, Alex Egger, Esaú García Sánchez-Torija, Thomas Günzel, Sebastian Di Luzio, Alexandru Obada, Maximilian Stadlmeier, Sebastian Voit, and Georg Carle. "The Case for Writing Network Drivers in High-Level Programming Languages". In: *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2019)*. IEEE. Cambridge, UK, Sept. 2019.

[44] Paul Emmerich, Maximilian Pudelko, Simon Bauer, Stefan Huber, Thomas Zwickl, and Georg Carle. "User Space Network Drivers". In: *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2019)*. IEEE. Cambridge, UK, Sept. 2019.

[45] David Flater and William F Guthrie. "A case study of performance degradation attributable to run-time bounds checks on C++ vector access". In: *Journal of research of the National Institute of Standards and Technology* 118 (2013), p. 260.

[46] Philip J Fleming and John J Wallace. "How not to lie with statistics: the correct way to summarize benchmark results". In: *Communications of the ACM* 29.3 (1986), pp. 218–221.

[47] Fluke networks. *Network Cable Propagation Delay*. `https://www.flukenetworks.com/knowledge-base/dtx-cableanalyzer/propagation-delay`.

[48] Alex Forencich, Alex C Snoeren, George Porter, and George Papen. "Corundum: An Open-Source 100-Gbps NIC". In: *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2020, pp. 38–46.

[49] Nate Foster, Nick McKeown, Jennifer Rexford, Guru Parulkar, Larry Peterson, and Oguz Sunay. "Using deep programmability to put network owners in control". In: *ACM SIGCOMM Computer Communication Review* 50.4 (2020), pp. 82–88.

[50] FreeBSD Project. "NETMAP(4)". In: *FreeBSD Kernel Interfaces Manual*. FreeBSD 11.1-RELEASE. 2017.

[51] Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle. "Comparison of Frameworks for High-Performance Packet IO". In: *Architectures for Networking and Communications Systems (ANCS)*. ACM. Oakland, CA, 2015, pp. 29–38.

[52] Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle. “Comparison of Frameworks for High-Performance Packet IO”. In: *Architectures for Networking and Communications Systems (ANCS)*. ACM. Oakland, CA, 2015.

[53] Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle. “Comparison of frameworks for high-performance packet IO”. In: *Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE/ACM. 2015.

[54] Jim Gettys and Kathleen Nichols. “Bufferbloat: Dark buffers in the internet”. In: *Queue* 9.11 (2011), p. 40.

[55] Gilberto Bertin. “XDP in practice: integrating XDP into our DDoS mitigation pipeline”. In: *Netdev 2.1, The Technical Conference on Linux Networking*. May 2017.

[56] Google. *Fuchsia git repositories*. `https://fuchsia.googlesource.com/`. 2019.

[57] Google. *Go - Frequently Asked Questions*. `https://go.dev/doc/faq`.

[58] Google. *Playing with QUIC*. `https://www.chromium.org/quic/playing-with-quic`. 2019.

[59] Luke Gorrie. *Snabb/ConnectX 100G transmit*. `https://rpubs.com/lukego/205901`.

[60] Isaac Gouy. *The Computer Language Benchmarks Game*. `https://benchmarksgame-team.pages.debian.net/benchmarksgame/`. 2019.

[61] Thomas Hallgren, Mark P Jones, Rebekah Leslie, and Andrew Tolmach. “A principled approach to operating system construction in Haskell”. In: *ACM SIGPLAN Notices*. Vol. 40. 9. ACM. 2005, pp. 116–128.

[62] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. “PacketShader: a GPU-accelerated software router”. In: *ACM SIGCOMM Computer Communication Review* 41.4 (2011), pp. 195–206.

[63] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. *PacketShader: Packet I/O Engine*. `https://shader.kaist.edu/packetshader/io_engine/`. 2010.

[64] HashiCorp. *Vagrant website*. `https://www.vagrantup.com/`. 2019.

[65] Galen Hunt, James R Larus, Martın Abadi, Mark Aiken, Paul Barham, Manuel Fahndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, et al. *An overview of the Singularity project*. Tech. rep. MSR-TR-2005-135, Microsoft Research, 2005.

[66] Stephen Ibanez, Muhammad Shahbaz, and Nick McKeown. “The Case for a Network Fast Path to the CPU”. In: *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*. 2019, pp. 52–59.

[67] IEEE. *IEEE 802.3-2018 IEEE Standard for Ethernet*. 2018.

[68] "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems". In: *IEEE 1588-2008* (2008). DOI: 10.1109/IEEESTD.2008.4579760.

[69] Intel. *DPDK Getting Started Guide for Linux*. http://dpdk.org/doc/guides/linux_gsg/sys_reqs.html.

[70] Intel. *Intel Server Adapters - Linux ixgbe Base Driver*. http://www.intel.com/support/network/adapter/pro100/sb/CS-032530.htm.

[71] Intel. *Network Function Framework for Go*. https://github.com/intel-go/nff-go. 2019.

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

[73] *Intel 82599 10 Gigabit Ethernet Controller Datasheet Rev. 2.9*. Intel Corporation. 2014.

[74] *Intel Advanced Encryption Standard (AES) New Instructions Set*. Intel Corporation. 2010.

[75] "Intel Data Direct I/O Technology (Intel DDIO): A Primer". In: (Feb. 2012). Available at https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology-brief.html.

[76] "Intel Ethernet Controller X540 Datasheet Rev. 2.7". In: Intel. 2014.

[77] "Intel Ethernet Controller XL710 Datasheet Rev. 2.1". In: Intel. 2014.

[78] *Intel i486 Microprocessor*. Intel Corporation. 1989.

[79] *Intel Xeon Processor D-1500 Product Family*. Intel Corporation. 2015.

[80] "Intel Xeon Processor E5-2600 v3 Product Family Architectural Overview". In: Intel. Nov. 2014.

[81] IO Visor Project. *BPF and XDP Features by Kernel Version*. https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md#xdp. 2019.

[82] IO Visor Project. *Introduction to XDP*. https://www.iovisor.org/technology/xdp. 2019.

[83] Jesús Leganés-Combarro et al. *Node-OS*. https://node-os.com. 2019.

[84] Jim Thompson. "DPDK, VPP & pfSense 3.0". In: *DPDK Summit Userspace*. Dublin, Ireland, Sept. 2017.

[85] Jonathan Corbet. "User-space networking with Snabb". In: *LWN.net*. Feb. 2017.

[86] Joseph Coffey. *Latency in optical fiber systems*. CommScope White Paper WP-111432. 2017.

[87] Michael Kerrisk. "mlock(2)". In: *Linux Programmer's Manual*. 2004.

[88] Richard B Kieburtz. *P-logic: Property verification for Haskell programs*. 2002.

[89] Joongi Kim, Seonggu Huh, Keon Jang, KyoungSoo Park, and Sue Moon. “The power of batching in the click modular router”. In: *Proceedings of the Asia-Pacific Workshop on Systems*. ACM. 2012.

[90] Kitura project. *Kitura website*. `https://www.kitura.io/`. 2019.

[91] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. “seL4: Formal Verification of an OS Kernel”. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009, pp. 207–220.

[92] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. “The Click Modular Router”. In: *ACM Transactions on Computer Systems* 18.3 (Aug. 2000), pp. 263–297. ISSN: 0734-2071. DOI: `10.1145/354871.354874`. URL: `http://doi.acm.org/10.1145/354871.354874`.

[93] Sameer G Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, KK Ramakrishnan, Timothy Wood, Mayutan Arumaithurai, and Xiaoming Fu. “NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains”. In: *Proceedings of the 2017 ACM SIGCOMM Conference*. 2016, pp. 71–84.

[94] D. Lacković and M. Tomić. “Performance analysis of virtualized VPN endpoints”. In: *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 2017, pp. 466–471. DOI: `10.23919/MIPRO.2017.7973470`.

[95] Linux documentation. *Linux kernel memory barriers*. `https://www.kernel.org/doc/Documentation/memory-barriers.txt`. 2019.

[96] Linux Foundation. *Data Plane Development Kit*. `http://dpdk.org`. 2013.

[97] Linux Kernel Documentation. *Page migration*. `https://www.kernel.org/doc/Documentation/vm/page_migration`.

[98] Linux Kernel Documentation. *VFIO - Virtual Function I/O*. `https://www.kernel.org/doc/Documentation/vfio.txt`. 2019.

[99] Luke Gorrie et al. *Snabb: Simple and fast packet networking*. `https://github.com/snabbco/snabb`. 2012.

[100] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. “Unikernels: Library Operating Systems for the Cloud”. In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’13. Houston, Texas, USA: ACM, 2013, pp. 461–472.

[101] Maximilian Pudelko. *ixy - DMA allocator on normal-sized pages*. `https://github.com/pudelkoM/ixy/tree/contiguous-pages`. 2019.

[102] Maximilian Pudelko. *ixy - head pointer writeback implementation*. https://github.com/pudelkoM/ixy/tree/head-pointer-writeback. 2019.

[103] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. "OpenFlow: enabling innovation in campus networks". In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74.

[104] *Mellanox BlueField SmartNIC for Ethernet*. Nvidia Corporation. 2020.

[105] Microsoft. *.NET Garbage Collection Latency Modes*. https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/latency. 2019.

[106] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. De Turck, and R. Boutaba. "Network Function Virtualization: State-of-the-Art and Research Challenges". In: *IEEE Communications Surveys & Tutorials* 18.1 (2016), pp. 236–262.

[107] MirageOS project. *Cstruct*. https://github.com/mirage/ocaml-cstruct. 2019.

[108] MirageOS project. *Performance harness for MirageOS 3*. https://github.com/mirage/mirage/issues/685. 2019.

[109] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Wolfgang E Nagel. "Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture". In: *2015 44th International Conference on Parallel Processing*. IEEE. 2015, pp. 739–748.

[110] *NetFPGA*. http://netfpga.org/.

[111] *netmap*. https://github.com/luigirizzo/netmap.

[112] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. "Understanding PCIe performance for end host networking". In: *SIGCOMM 2018*. ACM. 2018, pp. 327–341.

[113] Node.js Foundation. *Node.js*. https://nodejs.org. 2019.

[114] ntop. *Introducing PF_RING DNA (Direct NIC Access)*. https://www.ntop.org/pf_ring/introducing-pf_ring-dna-direct-nic-access/. 2010.

[115] ntop. *PF_RING ZC (Zero Copy)*. http://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/.

[116] *ntop official website*. http://www.ntop.org/.

[117] OASIS VIRTIO TC. *Virtual I/O Device (VIRTIO) Version 1.0*. http://docs.oasis-open.org/virtio/virtio/v1.0/virtio-v1.0.pdf. Mar. 2016.

[118] OCaml Manual. *Optimisation with Flambda*. https://caml.inria.fr/pub/docs/manual-ocaml/flambda.html. 2019.

[119] Open vSwitch project. *Open vSwitch releases*. http://docs.openvswitch.org/en/latest/faq/releases/. 2019.

[120] OpenJDK. *Shenandoah GC*. https://wiki.openjdk.java.net/display/shenandoah/Main. 2019.

[121] *OPNVF Website - Vswitch Project Proposal*. https://wiki.opnfv.org/display/vsperf/Vswitch+Project+Proposal.

[122] Srivats P. *ostinato*. http://ostinato.org/.

[123] Mike Pall. *LuaJIT*. http://luajit.org/. 2020.

[124] Mike Pall. *LuaJIT FFI Library*. http://luajit.org/ext_ffi.html. 2020.

[125] Mike Pall. *LuaJIT in realtime applications*. http://www.freelists.org/post/luajit/LuaJIT-in-realtime-applications,3. Mailing list post. 2012.

[126] J. F. Palmer. "The Intel 8087 numeric data processor". In: *International Workshop on Managing Requirements Knowledge*. Los Alamitos, CA, USA: IEEE, 1980, p. 887. DOI: 10.1109/AFIPS.1980.108.

[127] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. "NetBricks: Taking the V out of NFV". In: *OSDI'16*. USENIX. 2016, pp. 203–216.

[128] Marcos Paredes-Farrera, Martin Fleury, and Mohammed 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.

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

[130] "PCI Express Base Specification Rev. 3.0". In: PCI-SIG. Nov. 2010.

[131] Larry L Peterson and Bruce S Davie. *Computer Networks: a systems approach*. 5th ed. Elsevier, 2012.

[132] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. "The Design and Implementation of Open vSwitch". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX, 2015, pp. 117–130. ISBN: 978-1-931971-218.

[133] *PFQ*. https://github.com/pfq/PFQ.

[134] pfSense project. *pfSense website*. https://www.pfsense.org/. 2019.

[135] *PF_RING ZC*. https://github.com/ntop/PF_RING.git.

[136] *Pktgen-DPDK*. http://dpdk.org/browse/apps/pktgen-dpdk/refs/.

[137] "Product Brief - Intel Ethernet Controller XL710 10/40 GbE". In: Intel. 2014.

[138] *Product Brief: AMD Ryzen Embedded V1000 Processor Family*. AMD. 2018.

[139] Project Zero. *Meltdown and Spectre*. https://meltdownattack.com/. 2018.

[140] Maximilian Pudelko, Paul Emmerich, Sebastian Gallenmüller, and Georg Carle. "Performance Analysis of VPN Gateways". In: *2020 IFIP Networking Conference (Networking)*. IFP. 2020, pp. 325–333.

[141] PyPi. *Download statistics for PyUSB*. `https://pypistats.org/packages/pyusb`. 2019.

[142] PyPi. *PyUSB package*. `https://pypi.org/project/pyusb/`. 2019.

[143] Redox developers. *Redox project page*. `https://www.redox-os.org/`. 2019.

[144] Rick Hudson. "Go GC: Latency Problem Solved". In: *GopherCon Denver* (July 2015).

[145] Dennis M Ritchie. "The development of the C language". In: *ACM Sigplan Notices* 28.3 (1993), pp. 201–208. DOI: 10.1145/155360.155580.

[146] Dennis M Ritchie and K. Thompson. "The UNIX Time-Sharing System". In: *Communications of the ACM* 17.7 (July 1974), pp. 365–375.

[147] Stuart Ritchie. "Systems programming in Java". In: *IEEE Micro* 17.3 (1997), pp. 30–35. ISSN: 0272-1732. DOI: 10.1109/40.591652.

[148] Luigi Rizzo. "netmap: A Novel Framework for Fast Packet I/O". In: *USENIX Annual Technical Conference*. 2012, pp. 101–112.

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

[150] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. "Beyond Softnet". In: *Proceedings of the 5th Annual Linux Showcase & Conference*. Vol. 5. 2001, pp. 18–18.

[151] Peter H. Salus. "Unix at 25". In: *BYTE magazine* 19.10 (Oct. 1994), 75pp.

[152] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. "PISCES: A Programmable, Protocol-independent Software Switch". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. 2016, pp. 525–538.

[153] Snabb Project. *Tuning the performance of the lwaftr*. `https://github.com/snabbco/snabb/blob/master/src/program/lwaftr/doc/performance.md`.

[154] Solarflare. *OpenOnload Website*. `http://www.openonload.org/`.

[155] Joel Sommers and Paul Barford. "Self-Configuring Network Traffic Generation". In: *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*. IMC '04. Taormina, Sicily, Italy: ACM, 2004, pp. 68–81. ISBN: 1-58113-821-0.

[156] W. Su, D. Cohen, J. N. Seizovic, A. E. Kulawik, R. E. Felderman, N. J. Boden, and C. L. Seitz. "Myrinet: A Gigabit-per-Second Local Area Network". In: *IEEE Micro* 15.01 (1995), pp. 29–36. ISSN: 1937-4143. DOI: 10.1109/40.342015.

[157] Sun microsystems and IBM. *JavaOS for Business Device Driver Guide*. https://www.oracle.com/technetwork/java/josddk-150086.pdf. June 1998.

[158] M Tahhan, B. O'Mahony, and Al Morton. *Benchmarking Virtual Switches in the Open Platform for NFV (OPNFV)*. RFC 8204 (Informational). Internet Engineering Task Force, 2017.

[159] TAPS Working Group. *An Abstract Application Layer Interface to Transport Services*. 2021.

[160] TechEmpower. *TechEmpower Framework Benchmarks*. https://github.com/TechEmpower/FrameworkBenchmarks. 2019.

[161] Gavin Thomas. *A proactive approach to more secure code*. https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/. 2019.

[162] *trafgen*. http://netsniff-ng.org/.

[163] Vapor project. *Vapor website*. https://vapor.codes/. 2019.

[164] Péter Vörös, Dániel Horpácsi, Róbert Kitlei, Dániel Leskó, Máté Tejfel, and Sándor Laki. "T4P4S: A Target-independent Compiler for Protocol-independent Packet Processors". In: *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE. 2018, pp. 1–8.

[165] Keith Wiles. *Pktgen-DPDK*. http://github.com/Pktgen/Pktgen-DPDK/.

[166] Yinglin Yang, Sudeep Goswami, and Carl G. Hansen. *10GBASE-T Ecosystem is Ready for Broad Adoption*. White paper. 2012.

[167] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. "StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs". In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, 2016, pp. 43–56. ISBN: 978-1-931971-30-0.

[168] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. "A Formally Verified NAT". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. 2017, pp. 141–154.

[169] Noa Zilberman, Andrew W Moore, and Jon A Crowcroft. "From photons to big-data applications: terminating terabits". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 374.2062 (2016), p. 20140445.

[170] Noa Zilberman, Yury Audzevich, G Adam Covington, and Andrew W Moore. "NetFPGA SUME: Toward 100 gbps as research commodity". In: *Micro* 5 (2014).

Appendices

Appendix A

MoonGen Example: Basic packet generator

A MoonGen script generating a configurable number of UDP flows at a given rate.

The script showcases the following MoonGen features:

- Randomized flows
- ARP lookups
- CSV statistics
- The embedded REST webserver to control MoonGen via HTTP/json

```
1  --- A simple UDP packet generator
2  local mg     = require "moongen"
3  local device = require "device"
4  local stats  = require "stats"
5  local log    = require "log"
6  local memory = require "memory"
7  local arp    = require "proto.arp"
8  local server = require "webserver"
9
10 -- set addresses here
11 local DST_MAC       = nil -- resolved via ARP on GW_IP or DST_IP, can be overriden with a string here
12 local PKT_LEN       = 60
13 local SRC_IP        = "10.0.0.10"
14 local DST_IP        = "10.1.0.10"
15 local SRC_PORT_BASE = 1234 -- actual port will be SRC_PORT_BASE * random(NUM_FLOWS)
16 local DST_PORT      = 1234
17 local NUM_FLOWS     = 1000
18 -- used as source IP to resolve GW_IP to DST_MAC
19 -- also respond to ARP queries on this IP
20 local ARP_IP = SRC_IP
21 -- used to resolve DST_MAC
```

```lua
local GW_IP  = DST_IP


-- the configure function is called on startup with a pre-initialized command line parser
function configure(parser)
    parser:description("Edit the source to modify constants like IPs and ports.")
    parser:argument("dev", "Devices to use."):args("+"):convert(tonumber)
    parser:option("-t --threads", "Number of threads per device."):args(1):convert(tonumber):default(1)
    parser:option("-r --rate", "Transmit rate in Mbit/s per device."):args(1)
    parser:option("-w --webserver", "Start a REST API on the given port."):convert(tonumber)
    parser:option("-o --output", "File to output statistics to")
    parser:option("-s --seconds", "Stop after n seconds")
    parser:flag("-a --arp", "Use ARP.")
    parser:flag("--csv", "Output in CSV format")
    return parser:parse()
end

function master(args,...)
    -- configure devices and queues
    local arpQueues = {}
    for i, dev in ipairs(args.dev) do
        -- arp needs extra queues
        local dev = device.config{
            port = dev,
            txQueues = args.threads + (args.arp and 1 or 0),
            rxQueues = args.arp and 2 or 1
        }
        args.dev[i] = dev
        if args.arp then
            table.insert(arpQueues, {
                rxQueue = dev:getRxQueue(1), txQueue = dev:getTxQueue(args.threads), ips = ARP_IP
            })
        end
    end
    device.waitForLinks()

    -- start ARP task and do ARP lookup (if not hardcoded above)
    if args.arp then
        arp.startArpTask(arpQueues)
        if not DST_MAC then
            log:info("Performing ARP lookup on %s, timeout 3 seconds.", GW_IP)
            DST_MAC = arp.blockingLookup(GW_IP, 3)
            if not DST_MAC then
                log:info("ARP lookup failed, using default destination mac address")
                DST_MAC = "01:23:45:67:89:ab"
            end
        end
        log:info("Destination mac: %s", DST_MAC)
    end

    if args.webserver then
        server.startWebserverTask{
            port = args.webserver
        }
```

```
76      end
77
78      -- print statistics
79      stats.startStatsTask{devices = args.dev, file = args.output, format = args.csv and "csv" or "plain"}
80
81      -- configure tx rates and start transmit slaves
82      for i, dev in ipairs(args.dev) do
83          for i = 1, args.threads do
84              local queue = dev:getTxQueue(i - 1)
85              if args.rate then
86                  queue:setRate(args.rate / args.threads)
87              end
88              mg.startTask("txSlave", queue, DST_MAC)
89          end
90      end
91
92      if args.seconds then
93          mg.setRuntime(tonumber(args.seconds))
94      end
95
96      mg.waitForTasks()
97  end
98
99  function txSlave(queue, dstMac)
100     -- memory pool with default values for all packets, this is our archetype
101     local mempool = memory.createMemPool(function(buf)
102         buf:getUdpPacket():fill{
103             -- fields not explicitly set here are initialized to reasonable defaults
104             ethSrc = queue, -- MAC of the tx device
105             ethDst = dstMac,
106             ip4Src = SRC_IP,
107             ip4Dst = DST_IP,
108             udpSrc = SRC_PORT,
109             udpDst = DST_PORT,
110             pktLength = PKT_LEN
111         }
112     end)
113     -- a bufArray is just a list of buffers from a mempool that is processed as a single batch
114     local bufs = mempool:bufArray()
115     while mg.running() do -- check if Ctrl+c was pressed
116         -- this actually allocates some buffers from the mempool the array is associated with
117         -- this has to be repeated for each send because sending is asynchronous,
118         -- we cannot reuse the old buffers here (async interface)
119         bufs:alloc(PKT_LEN)
120         for i, buf in ipairs(bufs) do
121             -- packet framework allows simple access to fields in complex protocol stacks
122             local pkt = buf:getUdpPacket()
123             pkt.udp:setSrcPort(SRC_PORT_BASE + math.random(0, NUM_FLOWS - 1))
124         end
125         -- UDP checksums are optional, so using just IPv4 checksums would be sufficient here
126         -- UDP checksum offloading is comparatively slow: NICs typically do not support calculating the
127         -- pseudo-header checksum so this is done in SW
128         bufs:offloadUdpChecksums()
129         -- send out all packets and frees old bufs that have been sent
```

```
130             queue:send(bufs)
131         end
132     end
```

Appendix B

MoonGen Example: Quality of Service Test

A MoonGen script generating two different UDP flows and reporting their respective latencies. This can be used validate if a device is properly applying quality of service metrics.

The script showcases the following MoonGen features:

- Multi-threading with multiple device queues
- Hardware rate control
- Hardware timestamping
- Custom statistics

```
1  --- This script implements a simple QoS test by generating two flows and measuring their latencies.
2  local mg     = require "moongen"
3  local memory = require "memory"
4  local device = require "device"
5  local ts     = require "timestamping"
6  local filter = require "filter"
7  local stats  = require "stats"
8  local hist   = require "histogram"
9  local timer  = require "timer"
10 local log    = require "log"
11
12 local PKT_SIZE    = 124 -- without CRC
13 -- hard-coded MAC addresses, see the generic pktgen example for getting this via ARP
14 local ETH_DST    = "10:11:12:13:14:15" -- src mac is taken from the NIC
15 local IP_SRC    = "192.168.0.1"
16 local NUM_FLOWS    = 256 -- src ip will be IP_SRC + random(0, NUM_FLOWS - 1)
17 local IP_DST    = "10.0.0.1"
18 local PORT_SRC    = 1234
19 local PORT_FG    = 42
```

```lua
local PORT_BG    = 43

function configure(parser)
    parser:description("Generates two flows of traffic and compares them." ..
        " This example requires an ixgbe NIC due to the used hardware features.")
    parser:argument("txDev", "Device to transmit from."):convert(tonumber)
    parser:argument("rxDev", "Device to receive from."):convert(tonumber)
    parser:option("-f --fg-rate", "Foreground traffic rate in Mbit/s."):default(1000)
        :convert(tonumber):target("fgRate")
    parser:option("-b --bg-rate", "Background traffic rate in Mbit/s."):default(4000)
        :convert(tonumber):target("bgRate")
end

function master(args)
    -- 3 tx queues: traffic, background traffic, and timestamped packets
    -- 2 rx queues: traffic and timestamped packets
    local txDev, rxDev
    -- these two cases could actually be merged as re-configurations of ports are ignored
    -- the dual-port case could just config the 'first' device with 2/3 queues
    -- however, this example scripts shows the explicit configuration instead of implicit magic
    if args.txDev == args.rxDev then
        -- sending and receiving from the same port
        txDev = device.config{port = args.txDev, rxQueues = 2, txQueues = 3}
        rxDev = txDev
    else
        -- two different ports, different configuration
        txDev = device.config{port = args.txDev, rxQueues = 1, txQueues = 3}
        rxDev = device.config{port = args.rxDev, rxQueues = 2}
    end
    -- wait until the links are up
    device.waitForLinks()
    log:info("Sending %d MBit/s background traffic to UDP port %d", args.bgRate, PORT_BG)
    log:info("Sending %d MBit/s foreground traffic to UDP port %d", args.fgRate, PORT_FG)
    -- setup rate limiters for CBR traffic
    -- see l2-poisson.lua for an example with different traffic patterns
    txDev:getTxQueue(0):setRate(args.bgRate)
    txDev:getTxQueue(1):setRate(args.fgRate)
    -- background traffic
    if args.bgRate > 0 then
        mg.startTask("loadSlave", txDev:getTxQueue(0), PORT_BG)
    end
    -- high priority traffic (different UDP port)
    if args.fgRate > 0 then
        mg.startTask("loadSlave", txDev:getTxQueue(1), PORT_FG)
    end
    -- count the incoming packets
    mg.startTask("counterSlave", rxDev:getRxQueue(0))
    -- measure latency from a second queue
    mg.startSharedTask("timerSlave", txDev:getTxQueue(2), rxDev:getRxQueue(1),
        PORT_BG, PORT_FG, args.fgRate / (args.fgRate + args.bgRate))
    -- wait until all tasks are finished
    mg.waitForTasks()
end

```

```lua
function loadSlave(queue, port)
    mg.sleepMillis(100) -- wait a few milliseconds to ensure that the rx thread is running
    local mem = memory.createMemPool(function(buf)
        buf:getUdpPacket():fill{
            pktLength = PKT_SIZE, -- this sets all length headers fields in all used protocols
            ethSrc = queue, -- get the src mac from the device
            ethDst = ETH_DST,
            -- ipSrc will be set later as it varies
            ip4Dst = IP_DST,
            udpSrc = PORT_SRC,
            udpDst = port,
            -- payload will be initialized to 0x00 as new memory pools are initially empty
        }
    end)
    local txCtr = stats:newManualTxCounter("Port " .. port, "plain")
    local baseIP = parseIPAddress(IP_SRC)
    -- a buf array is essentially a very thing wrapper around a
    -- rte_mbuf*[], i.e. an array of pointers to packet buffers
    local bufs = mem:bufArray()
    while mg.running() do
        -- allocate buffers from the mem pool and store them in this array
        bufs:alloc(PKT_SIZE)
        for _, buf in ipairs(bufs) do
            -- modify some fields here
            local pkt = buf:getUdpPacket()
            -- select a randomized source IP address
            -- you can also use a wrapping counter instead of random
            pkt.ip4.src:set(baseIP + math.random(NUM_FLOWS) - 1)
            -- you can modify other fields here (e.g. different source ports or destination addresses)
        end
        -- send packets
        bufs:offloadUdpChecksums()
        txCtr:updateWithSize(queue:send(bufs), PKT_SIZE)
    end
    txCtr:finalize()
end

function counterSlave(queue)
    -- the simplest way to count packets is by receiving them all
    -- an alternative would be using flow director to filter packets by port and use the queue statistics
    local bufs = memory.bufArray()
    local ctrs = {}
    while mg.running(100) do
        local rx = queue:recv(bufs)
        for i = 1, rx do
            local buf = bufs[i]
            local pkt = buf:getUdpPacket()
            local port = pkt.udp:getDstPort()
            local ctr = ctrs[port]
            if not ctr then
                ctr = stats:newPktRxCounter("Port " .. port, "plain")
                ctrs[port] = ctr
            end
            ctr:countPacket(buf)
```

```
128         end
129         -- update() on rxPktCounters must be called to print statistics periodically
130         -- this is not done in countPacket() for performance reasons (needs to check timestamps)
131         for k, v in pairs(ctrs) do
132             v:update()
133         end
134         bufs:freeAll()
135     end
136     for k, v in pairs(ctrs) do
137         v:finalize()
138     end
139 end
140 
141 
142 function timerSlave(txQueue, rxQueue, bgPort, port, ratio)
143     local txDev = txQueue.dev
144     local rxDev = rxQueue.dev
145     local timestamper = ts:newUdpTimestamper(txQueue, rxQueue)
146     local histBg, histFg = hist(), hist()
147     -- wait one second, otherwise we might start timestamping before the load is applied
148     mg.sleepMillis(1000)
149     local baseIP = parseIPAddress(IP_SRC)
150     local rateLimit = timer:new(0.001)
151     while mg.running() do
152         local port = math.random() <= ratio and port or bgPort
153         local lat = timestamper:measureLatency(PKT_SIZE, function(buf)
154             local pkt = buf:getUdpPacket()
155             pkt:fill{
156                     -- this sets all length headers fields in all used protocols
157                 pktLength = PKT_SIZE,
158                 ethSrc = txQueue, -- get the src mac from the device
159                 ethDst = ETH_DST,
160                 -- ipSrc will be set later as it varies
161                 ip4Dst = IP_DST,
162                 udpSrc = PORT_SRC,
163                 udpDst = port,
164             }
165             pkt.ip4.src:set(baseIP + math.random(NUM_FLOWS) - 1)
166         end)
167         if lat then
168             if port == bgPort then
169                 histBg:update(lat)
170             else
171                 histFg:update(lat)
172             end
173         end
174         rateLimit:wait()
175         rateLimit:reset()
176     end
177     mg.sleepMillis(100) -- to prevent overlapping stdout
178     histBg:save("hist-background.csv")
179     histFg:save("hist-foreground.csv")
180     histBg:print("Background traffic")
```

```
181       histFg:print("Foreground traffic")
182   end
```